Software Engineering Institute

Secure Design Patterns

Chad Dougherty
Kirk Sayre
Robert C. Seacord
David Svoboda
Kazuya Togashi

March 2009

TECHNICAL REPORT
CMU/SEI-2009-TR-010
ESC-TR-2009-010

CERT Program

http://www.cert.org/

Carnegie Mellon

Table of Contents

List of Figures

List of Tables

Acknowledgments

Thanks to our sponsor, JPCERT/CC. Thanks to our SEI editors Pamela Curtis and Amanda Parente, and to Yurie Ito and Bob Rosenstein for making this work possible.

Abstract

The cost of fixing system vulnerabilities and the risk associated with vulnerabilities after system deployment are high for both developers and end users. While there are a number of best practices available to address the issue of software security vulnerabilities, these practices are often difficult to reuse due to the implementation-specific nature of the best practices. In addition, greater understanding of the root causes of security flaws has led to a greater appreciation of the importance of taking security into account in all phases in the software development life cycle, not just in the implementation and deployment phases. This report describes a set of *secure design patterns*, which are descriptions or templates describing a general solution to a security problem that can be applied in many different situations. Rather than focus on the implementation of specific security mechanisms, the secure design patterns detailed in this report are meant to eliminate the accidental insertion of vulnerabilities into code or to mitigate the consequences of vulnerabilities. The patterns were derived by generalizing existing best security design practices and by extending existing design patterns with security-specific functionality. They are categorized according to their level of abstraction: architecture, design, or implementation.

1 Introduction

1.1 About Secure Design Patterns

A pattern is a general reusable solution to a commonly occurring problem in design. Note that a design pattern is not a finished design that can be transformed directly into code. It is a description or template for how to solve a problem that can be used in many different situations. Algorithms are not thought of as design patterns because they solve computational problems rather than design problems.

Secure design patterns are meant to eliminate the accidental insertion of vulnerabilities into code and to mitigate the consequences of these vulnerabilities. In contrast to the design-level patterns popularized in [Gamma 1995], secure design patterns address security issues at widely varying levels of specificity ranging from architectural-level patterns involving the high-level design of the system down to implementation-level patterns providing guidance on how to implement portions of functions or methods in the system.

1.1.1 Pattern History

1977/79 – Architect Christopher Alexander introduced the concept of design patterns with respect to the design of buildings and towns [Alexander 1977].

1987 – Beck and Cunningham experimented with applying patterns to programming and presented at OOPSLA [Beck 1987].

1994/95 – The “Gang of Four” (Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides) published a book containing a large number of design-level patterns aimed at object oriented programming languages [Gamma 1995].

1997 – Yoder and Baraclow published a paper outlining several security patterns [Yoder 1997].

1.1.2 Resources

A significant amount of research has already been performed in the field of security patterns. This section lists some of the major contributions to the field and provides a brief description of each piece of work.

- *Security Design Patterns, Part 1* [Romanosky 2001]. The patterns in this report address high-level security concerns, such as how to handle communication with untrusted third-party systems and the importance of multi-layered security. In addition, the patterns in this report address high-level process issues such as the use of white-hat penetration testing and addressing simple, high-impact security issues early in the system development and configuration process.
- *Core Security Patterns Book* [Steel 2005]. This book concentrates on security patterns for J2SE, J2EE, J2ME, and Java Card platform applications. The patterns contained in this book are generally design-level patterns applicable primarily to Java web applications.

- *Security Patterns: Integrating Security and Systems Engineering* [Schumacher 2006]. This book contains a large number of patterns at varying levels of specificity. The patterns in this book range from high-level patterns involving the processes used to develop secure systems to design-level patterns addressing how to create objects with different access privileges.
- *Open Group Guide to Security Patterns* [Blakely 2004]. This report contains architectural-level patterns and design-level patterns focusing on system availability and the protection of privileged resources. The patterns presented in this report are general patterns applicable to systems programmed in many different languages.
- *Security Patterns Repository* [Kienzle 2003]. This report contains both design-level patterns applicable to designing and building secure applications and procedural patterns that are applicable to the process of designing, building, and configuring secure applications.

1.2 Purpose

1.2.1 Problem to Be Solved

The cost of fixing system vulnerabilities and the risk associated with vulnerabilities after system deployment are high for both developers and end users. Steps to reduce the cost of system maintenance and the risk of security vulnerabilities need to be adopted by software development organizations. While there are a number of best practices available to address the issue of software security vulnerabilities, these practices are frequently difficult to reuse due to the implementation-specific nature of the best practices. In addition, greater understanding of the root causes of security flaws has led to a greater appreciation of the importance of taking security into account in all phases in the software development life cycle, not just in the implementation and deployment phases. Many current best security practices focus on implementation and deployment issues and so do not address security flaws introduced in earlier phases of the development process.

Various secure design patterns detailed in this report address security issues in the architectural design, detailed design, and implementation phases of the software development life cycle. In addition, several of the presented patterns were created by analyzing and generalizing existing, proven best practices. Some potential new secure design patterns, created by extending existing design patterns to take security issues into account, are also proposed in this report.

The creation of secure design patterns by generalizing and cataloging existing best practices and by the extension of existing non-secure design patterns benefits the developers of secure software products. By using reusable security patterns, developers can reduce the cost associated with producing secure products while at the same time reducing the cost and the risk associated with security vulnerabilities for both developers and end users.

1.2.2 Approach

The approach taken to define the patterns in this document is to

- capture a number of demonstrably security-effective techniques from existing designs that can and should be replicated in other systems
- distill and document these techniques as secure design patterns

Additionally, several new but unproven patterns are proposed in this document. These patterns are secure extensions of some well-known patterns described in [Gamma 1995].

Inspirations for the patterns in this document include

- OpenBSD-derived projects
- qmail and Postfix mail system designs
- relevant recommendations from Kernighan and Pike's *The Practice of Programming* [Kernighan 1999]
- well-known basic design patterns from [Gamma 1995]

1.2.3 Intended Audience

The intended audience of this report is software engineers producing software artifacts at varying levels of abstraction, including architecture, design, and implementation.

The secure patterns in this report are grouped accordingly.

1.3 Scope

Secure design patterns, as described by this report, provide general design guidance to eliminate the introduction of vulnerabilities into code or mitigate the consequences of vulnerabilities. Secure design patterns are not restricted to object-oriented design approaches but may also be applied, in many cases, to procedural languages. These patterns are at a higher level of abstraction than secure coding guidelines.

Secure design patterns differ from *security patterns* in that they do not describe specific security mechanisms (such as access control, authentication, and authorization (AAA) and logging), define secure development processes, or provide guidance on the configuration of existing secure systems.

Three general classes of patterns are presented in this document:

- **Architectural-level patterns**. Architectural-level patterns focus on the high-level allocation of responsibilities between different components of the system and define the interaction between those high-level components. The architectural-level patterns defined in this document are
  - Distrustful Decomposition
  - PrivSep (Privilege Separation)
  - Defer to Kernel
- **Design-level patterns**. Design-level patterns describe how to design and implement pieces of a high-level system component, that is, they address problems in the internal design of a single high-level component, not the definition and interaction of high-level components themselves. The design-level patterns defined in this document are
  - Secure State Machine
  - Secure Visitor

- **Implementation-level patterns**. Implementation-level patterns address low-level security issues. Patterns in this class are usually applicable to the implementation of specific functions or methods in the system. Implementation-level patterns address the same problem set addressed by the CERT Secure Coding Standards [CERT 2009a] and are often linked to a corresponding secure coding guideline. Implementation-level patterns defined in this document are
  - Secure Directory
  - Pathname Canonicalization
  - Input Validation
  - Runtime Acquisition Is Initialization

This report does not provide a complete secure design pattern catalog. In the creation of this report, some, but by no means all, best practices used in the creation of secure software were analyzed and generalized. Future work will extend the catalog of secure design patterns.

1.4 Format and Conventions

The template for describing design patterns used in [Gamma 1995] was used to describe the secure design patterns in this report. The sections in the template are shown in Table 1. Sections whose names are italicized are optional.

Table 1: Pattern Elements

Element	**Description**
Intent	The problem solved by the design pattern and its general rationale and purpose.
Also Known As	Other names for the pattern, if any are known.
Example	A real-world example demonstrating the existence of the problem and the need for the pattern. Throughout the description, we refer to examples to illustrate solutions and implementation aspects, where this is necessary or useful.
Motivation	A description of situations in which the pattern may apply and a more detailed description of the problem that the pattern is intended to solve.
Applicability	A general description of the characteristics a program must have for the pattern to be useful in the design or implementation of the program.
Structure	A textual or graphical description of the relationship between the various participants in the pattern. This provides a detailed specification of the structural aspects of the pattern, using appropriate notations.
Participants	The entities involved in the pattern.
Consequences	The benefits the pattern provides and any potential liabilities.
Implementation	Guidelines for implementing the pattern. These are only a suggestion, not an immutable rule. You should adapt the implementation to meet your needs by adding different, extra, or more detailed steps or by reordering the steps. Whenever applicable we give UML fragments to illustrate a possible implementation, often describing details of the example problem.
Sample Code	Code providing an example of how to implement the pattern.
Example Resolved	An example of how the real-world example problem described in the *Example* section may be resolved through the use of the secure design pattern.
Known Uses	Examples of the use of the pattern, taken from existing systems.

2 The Architectural-Level Patterns

2.1 Distrustful Decomposition

2.1.1 Intent

The intent of the Distrustful Decomposition secure design pattern is to move separate functions into mutually untrusting programs, thereby reducing the

- attack surface of the individual programs that make up the system
- functionality and data exposed to an attacker if one of the mutually untrusting programs is compromised

2.1.2 Also Known As

Privilege reduction

2.1.3 Motivation

Many attacks target vulnerable applications running with elevated permissions. This allows the attacker to access more information and/or allows the attacker to perform more damage after exploiting a security hole in the application than if the application had been running with more restrictive permissions. Some examples of this class of attack are

- various attacks in which Internet Explorer running in an account with administrator privileges is compromised
- security flaws in Norton AntiVirus 2005 that allow attackers to run arbitrary VBS scripts when running with administrator privileges
- a buffer overflow vulnerability in BSD-derived telnet daemons that allows an attacker to run arbitrary code as root

All of these attacks take advantage of security flaw(s) in an application running with elevated privileges (root under UNIX or administrator under Windows) to compromise the application and then use the application's elevated privileges and basic functionality to compromise other applications running on the computer or to access sensitive data. The Distrustful Decomposition pattern isolates security vulnerabilities to a small subset of a system such that compromising a single component of the system does not lead to the entire system being compromised. The attacker will only have the functionality and data of the single compromised component at their disposal for malicious activity, not the functionality and data of the entire application.

2.1.4 Applicability

This pattern applies to systems where files or user-supplied data must be handled in a number of different ways by programs running with varying privileges and responsibilities. A naive implementation of this system may allocate many disparate functions to the same program, forcing the program to be run at the privilege level required by the program function requiring the highest privilege level. This provides a large attack surface for attackers and leaves an attacker with access to a system with a high privilege level if the system is compromised.

A system can make use of the Distrustful Decomposition pattern if

- the system performs more than one high-level function
- the various functions of the system require different privilege levels

2.1.5 Structure

The general structure of this pattern breaks the system up into two or more programs that run as separate processes, with each process potentially having different privileges. Each process handles a small, well-defined subset of the system functionality. Communication between processes occurs using an inter-process communication mechanism such as RPC, sockets, SOAP, or shared files.

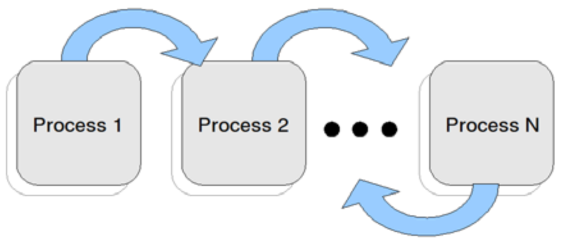

Figure 1: General Structure of the Distrustful Decomposition Secure Design Pattern

2.1.6 Participants

These are the participants in the Distrustful Decomposition pattern:

- a number of separate programs, each running in a separate process. For more complete separation, each process could have a unique user ID that does not share any privileges with the other user IDs.
- a local user or a remote system connecting over a network
- possibly the system's file system
- possibly an inter-process communication mechanism such as UNIX domain sockets, RPC, or SOAP

2.1.7 Consequences

Distrustful Decomposition prevents an attacker from compromising an entire system in the event that a single component program is successfully exploited because no other program trusts the results from the compromised one.

2.1.8 Implementation

This pattern employs nothing beyond the standard process/privilege model already existing in the operating system. Each program runs in its own process space with potentially separate user privileges. Communication between separate programs is either one-way or two-way.

- One-way. Only `fork()`/`exec()` (UNIX/Linux/etc.), `CreateProcess()` (Windows Vista), or some other OS-specific method of programmatic process creation is used to transfer control. One-way communication reduces the coupling between processes, making it more difficult for an attacker to compromise one system component from another, already compromised component.

- Two-way. A two-way inter-process communication mechanism like TCP or SOAP is used. Extra care must be taken when using a two-way communication mechanism because it is possible for one process involved in the two-way communication to be compromised and under the control of an attacker. As with the file system, two-way communication should not be inherently trusted.

The file system may be a means of interaction, but no component places any inherent trust in the contents of the file.

2.1.9 Sample Code

An excellent example system where this pattern is applied is the qmail mail system, which is a complex system with a large combination of interactions between systems, users, and software components.

The overall structure of the qmail system is shown in Figure 2 [Oppermann 1998].

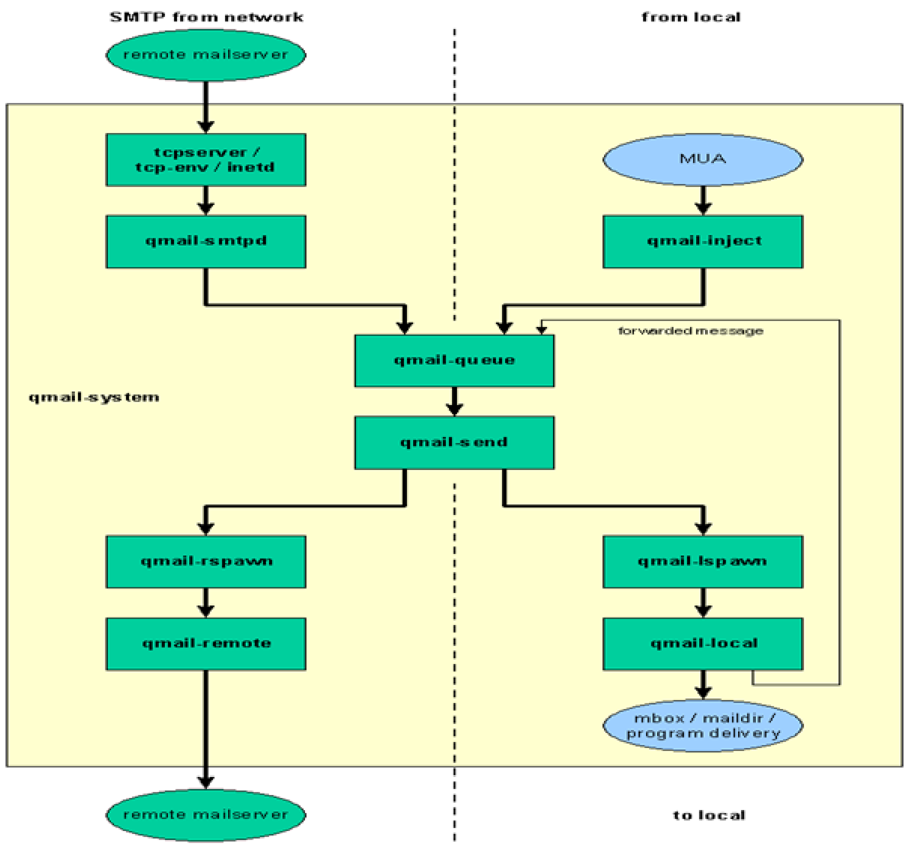

Figure 2: Structure of the Qmail Mail System[1]

1 Source: http://www.nrg4u.com/qmail/the-big-qmail-picture-103-p1.gif. Used with permission from the author.

The actual source code for the qmail system is omitted here; see the qmail website [Bernstein 2008] for examples.

2.1.10 Known Uses

- The qmail mail system [Bernstein 2008].
- The Postfix mail system uses a similar pattern [Postfix].
- Microsoft mentions this general pattern when discussing how to run applications with administrator privileges [MSDN 2009b].

Distrustful decomposition for Windows Vista applications using user account control (UAC) is explicitly addressed in [Massa 2008].

2.2 PrivSep (Privilege Separation)

2.2.1 Intent

The intent of the PrivSep pattern is to reduce the amount of code that runs with special privilege without affecting or limiting the functionality of the program. The PrivSep pattern is a more specific instance of the Distrustful Decomposition pattern.

2.2.2 Motivation

In many applications, a small set of simple operations require elevated privileges, while a much larger set of complex and security error-prone operations can run in the context of a normal unprivileged user. For a more detailed discussion of the motivation for using this pattern, please see the motivation for the more general Distrustful Decomposition pattern.

Figure 3 provides a detailed view of a system where the PrivSep pattern could be applied and the security problems that can occur if the PrivSep pattern is not used [Provos 2003]. An implementation of `ftpd` is used as an example.

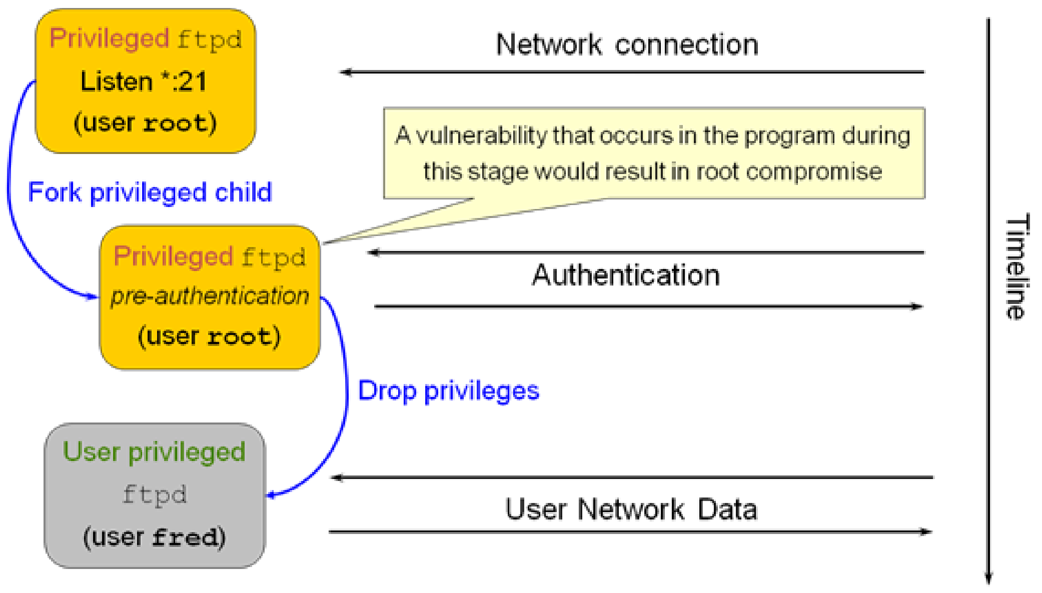

Figure 3: Vulnerable `ftpd` Program

The security flaw occurs when the privileged server establishes a connection with the as-yet untrusted system user and attempts to authenticate the user with a child possessing the same elevated privileges as the server. A malicious user could at this point exploit security holes in the privileged child and gain control of or access to a process with elevated privileges.

2.2.3 Applicability

In general, this pattern is applicable if the system performs a set of functions that

- do *not* require elevated privileges
- have relatively large attack surfaces in that the functions
  - have significant communication with untrusted sources
  - make use of complex, potentially error-prone algorithms

In particular, this pattern is especially useful for system services that must authenticate users and then allow the users to run interactive programs with normal, user-level privileges. It may be also be useful for other authenticating services.

2.2.4 Structure

Figure 4 shows the structure and behavior of the PrivSep pattern. Note that this diagram makes reference to the UNIX `fork()` function for creating child processes. When implementing the PrivSep pattern in a non-UNIX-based OS, a different, OS-specific function would be used in place of `fork()`. For example, under various versions of Windows, the `CreateProcess()` function is used to spawn a child process.

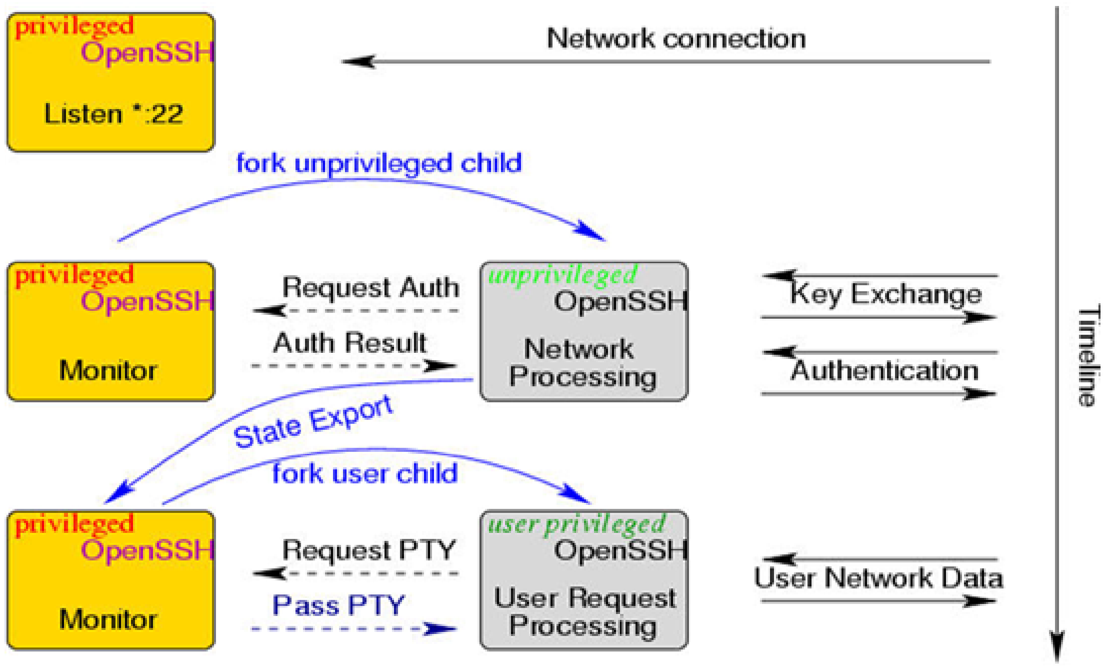

Figure 4: OpenSSH PrivSep Implementation[2]

[2] Source: http://www.citi.umich.edu/u/provos/ssh/priv.jpg.

2.2.5 Participants

Privileged Server Process

The privileged server process is responsible for fielding the initial requests for functionality that will eventually be handled by a child process with non-elevated privileges. The privileged server has an associated privileged userid (often `root` under UNIX-derived OSs, or `administrator` under various versions of Windows).

System User

The system user asks the system to perform some action. This initial request for functionality is directed by the user to the privileged server. The user can be local or remote. The user can communicate with the privileged server via an inter-process communication mechanism such as sockets or SOAP.

Unprivileged Client Process

The unprivileged client is responsible for handling the authentication of the user's request. Because it is not yet known if this is a valid request from a trusted user, the privileges of the child process handling authentication are limited as follows:

- The child process is given the minimal set of privileges allowed by the host OS. Under the UNIX privilege model, this is implemented by setting the user ID (UID) of the process to an unprivileged user ID.
- The root directory of the child process is set to an unimportant, empty directory or a jail [Seacord 2008]. This prevents the untrusted child process from accessing any of the files on the machine running the untrusted child.

User-Privileged Client Process

Once the system user and their request have been authorized, a child process with appropriate user-level privileges is spawned from the privileged server. The user-privileged child process actually handles the system user's request. The user-privileged child has its UID set to a local user ID.

2.2.6 Consequences

An adversary who gains control over the child

- is confined in its protection domain and does not gain control over the parent
- does not gain control of a process possessing elevated privileges, thereby limiting the damage that the adversary can inflict

Additional verification, such as code reviews, additional testing, and formal verification techniques, can be focused on code that is executed with special privilege, which can further reduce the incidence of unauthorized privilege escalation.

System administration overhead is usually increased to accommodate the management of new unprivileged user IDs.

2.2.7 Implementation

The PrivSep pattern consists of two phases, pre-authentication and post-authentication.

- Pre-authentication. A user has contacted a system service but is not yet authenticated; the unprivileged child has no process privileges and no rights to access the file system.

  The pre-authentication stage is implemented using two entities: a privileged parent process that acts as the monitor and an unprivileged child process that acts as the slave. The privileged parent can be modeled by a finite-state machine (FSM) that monitors the progress of the unprivileged child.
- Post-authentication. The user has successfully authenticated to the system. The child has the privileges of the user, including file system access, but does not hold any other special privilege.

The general process implemented in the PrivSep pattern is as follows:

1. Create a privileged server. Initial user requests will be directed to this server.
2. When a user request arrives at the server, the server will spawn off an untrusted, unprivileged child to handle the user interaction required during the authentication process.
3. After the user has been authenticated, the server will spawn off another child process with the appropriate UID to actually handle the user's request.

The unprivileged child is created by changing its UID or group ID (GID) to otherwise unused IDs. This is achieved by first starting a privileged monitor process that forks a slave process. To prevent access to the file system, the untrusted child changes the root of its file system to an empty directory in which no files may be written. The untrusted child process changes its UID or GID to the UID of an unprivileged user so as to lose its process privileges.

Slave requests to the monitor are performed using a standard inter-process communication mechanism.

2.2.8 Sample Code

A simple implementation of the PrivSep pattern using `fork()`, `chroot()`, and `setuid()` under Linux is as follows.

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <fcntl.h>
#include <stdlib.h>

// Define an unused UID.
#define UNPRIVILEGED_UID 123456789

// Hardcode the UID of the user. In reality the UID should not be
// hard coded.
#define USER_UID 1000
```

```
// The location of the empty directory to use as the root directory
// for the untrusted child process.
#define EMPTY_ROOT_DIR "/home/sayre/empty_dir"

/**
 * This defines the behavior for the spawned child, both the one with
 * no privileges and the one with user privileges.
 *
 * The parameters are:
 *
 * childUid - The UID to which to assign the spawned child.
 *
 * sock - The socket the child process will use for communication with
 * the privileged parent.
 */
void handleChild(uid_t childUid, int sock) {

  // A buffer to read in messages from the socket.
  char buffer[100];

  // Change the root of the untrusted child’s file system to an empty
  // directory, if we are the untrusted child.
  if (childUid != USER_UID) {
    if (chroot(EMPTY_ROOT_DIR) != 0) {
      printf("Cannot change root directory to %s.\n", EMPTY_ROOT_DIR);
      exit(7);
    }
  }
  // Immediately set the UID of the child to the user or
  // unprivileged UID.
  if (setuid(childUid) < 0) {
    printf("Cannot set UID to %d\n", childUid);
    exit(6);
  }

  // At this point the child no longer has the full privileges of the
  // privileged parent.

  // Are we the unprivileged child that is used to check
  // authorizations?
  if (childUid != USER_UID) {

    // Yes, we are the unprivileged child.

    // Ask the privileged parent to verify the credentials of the
    // child. Note that for the purposes of this simple example code
    // the "credentials" are represented very simply. In a real
    // application of the PrivSep pattern the credentials would be
    // handled in a much more robust fashion.
    send(sock, "VERIFY: MY_CREDS", 17, 0) ;

    // Read in the credential verification results from the privileged
    // parent.
    int size = recv(sock, buffer, sizeof(buffer), 0) ;

    // Was there an error reading the verification results from the
    // parent?
    if (size < 0) {
      printf("Read error in parent.\n");
      exit(2);
    }

    // Make sure the results string we have been sent is null
    // terminated.
    buffer[sizeof(buffer)-1] = '\0';

    // Were our credentials "authenticated"?
    if (strcmp("yes", buffer) != 0) {

      // Authorization denied. Kill the child process with an
      // appropriate error code.
      printf("Authorization denied.\n");
      exit(5);
    }

    // Our credentials were authorized.
    printf("Authorization approved.\n");

    // The unprivileged child now terminates. The privileged parent
    // will now spawn a child with user privileges.
    exit(0);
  }

  // We are the child with the user's UID. Our authorization has
  // already been approved.
  else {
    // Do the actual work of the verified child here...
    // ...
    // ...
    // ...
  }
}

int main(int argc, const char* argv[]) {

  // Create the socket pair that the parent and child will use to
  // communicate.
  int sockets[2];
  if (socketpair(PF_UNIX, SOCK_STREAM, AF_LOCAL, sockets) != 0) {

    // Creating the socket pair failed. Terminate the process.
    exit(1);
  }

  // A buffer to read in messages from the socket.
```

```
char buffer[100];

// Initially the spawned child should change its UID to an ID with
// no privileges.
uid_t childUid = UNPRIVILEGED_UID;

// Fork into a parent and an unprivileged child process.
pid_t pID = fork();

// Am I the child?
if (pID == 0) {
  // Use an unprivileged child to do the authorization.
  handleChild(childUid, sockets[0]);
}

// Did the fork fail?
else if (pID < 0) {
  printf("Fork failed\n");
  exit(3);
}

// I am the parent.
else {

  // As this point the parent expects the untrusted child to try to
  // get authorized.

  // Get the socket for the parent process.
  int sock = sockets[1];

  // Receive an authorization request from the child.
  int size = recv(sock, buffer, sizeof(buffer), 0) ;

  // Was there an error reading the authorization request message?
  if (size < 0) {
    printf("Read error in parent.\n");
    exit(4);
  }

  // Make sure the string we have been sent is null terminated.
  buffer[sizeof(buffer)-1] = '\0';

  // Do the "authorization" of the child. Note that in this simple
  // example the authorization process has been trivialized. In a
  // real application of the PrivSep pattern a much more robust
  // authorization process would be used.
  if (strcmp("VERIFY: MY_CREDS", buffer) == 0) {

    // Authorization succeeded. Tell the child.
    send(sock, "yes", 4, 0);

    // Because the "authorization" succeeded, spawn off a new child
    // with the user's UID that will do the real work.
```

```
      childUid = USER_UID;

      // Fork into a parent and an unprivileged child process.
      pid_t pID = fork();

      // Am I the child?
      if (pID == 0) {
        handleChild(childUid, sockets[0]);
      }

      // Did the fork fail?
      else if (pID < 0) {
        printf("Fork failed\n");
        exit(3);
      }

      // I am the parent.
      else {

        // Do some other parent operations, if needed...
        // ...
        // ...
        // ...
      }
    }
    else {
      // Authorization failed. Tell the child.
      send(sock, "no", 4, 0) ;
    }
  }
}
```

2.2.9 Known Uses

OpenBSD: `sshd`, `bgpd`/`ospfd`/`ripd`/`rtadvd`, X window server, `snmpd`, `ntpd`, `dhclient`, `tcpdump`, etc.

2.3 Defer to Kernel

2.3.1 Intent

The intent of this pattern is to clearly separate functionality that requires elevated privileges from functionality that does not require elevated privileges and to take advantage of existing user verification functionality available at the kernel level. Using existing user verification kernel functionality leverages the kernel's established role in arbitrating security decisions rather than reinventing the means to arbitrate security decisions at the user level.

The Defer to Kernel pattern is a specialization of the following patterns:

- CERT's Distrustful Decomposition secure design pattern
- the Reference Monitor security pattern by Schumacher et al.

> The Reference Monitor is a general pattern that describes how to define an abstract process that intercepts all requests for resources and checks them for compliance with authorizations [Schumacher 2006].

The primary difference between the Defer to Kernel pattern and the Reference Monitor pattern is that the Defer to Kernel pattern focuses on the use of user verification functionality provided by the OS kernel, whereas the Reference Monitor pattern does not specify the authorization method.

2.3.2 Motivation

A primary motivation for this pattern is to reduce or avoid the need for user programs that run with elevated privileges and are consequently susceptible to privilege escalation attacks. In UNIX-based systems, this means the reduction or avoidance of `setuid` programs. Under Windows, this means the avoidance of user programs running as administrator.

In addition, this pattern focuses on the reuse of user verification functionality provided by the OS kernel. The reuse of existing kernel functionality to verify users has these advantages:

- Developers do not have to write their own user identification and verification functionality.
- Testing and validation has already been performed on the existing kernel user identification and verification functionality.
- It is a more portable solution because it allows each OS to verify users in a manner consistent with each platform.

For a more detailed discussion of the motivation for using this pattern, please see the motivation for the more general Distrustful Decomposition pattern.

2.3.3 Applicability

The Defer to Kernel pattern is applicable if the system has the following characteristics:

- The system is run by users who do not have elevated privileges.
- Some (possibly all) of the functionality of the system requires elevated privileges.
- Prior to executing functionality that requires elevated privileges, the system must verify that the current user is allowed to execute the functionality.

In particular, for systems running on UNIX-based operating systems, the Defer to Kernel pattern is applicable if the system has the following characteristics:

- The program must run under a special UID to perform some or all of its tasks.
- The program accepts files or job requests submitted by users.
- For local users, the program needs to know which UID or GID submitted each file or job request, for access control or for accounting.
- For non-local users, the program uses some other user verification and logging mechanism.

2.3.4 Structure

The Defer to Kernel pattern implements a basic client-server architecture. The server runs with elevated privileges, accepts user job requests from clients, and, when possible, uses existing kernel functionality to verify users.

The general structure of the Defer to Kernel pattern is shown in Figure 5.

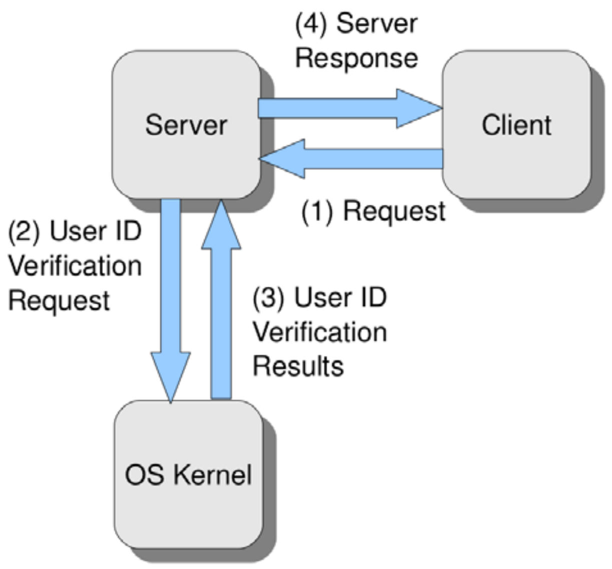
(4) Server Response
Server
Client
(1) Request
(2) User ID Verification Request
(3) User ID Verification Results
OS Kernel

Figure 5: General Structure of the Defer to Kernel Pattern

Figure 6 shows the structure of the Defer to Kernel pattern when the system has the following characteristics:

- The system is implemented under a UNIX-based OS.
- The system uses `getpeereid()` for user verification.
- The server only accepts files and job requests from local users.

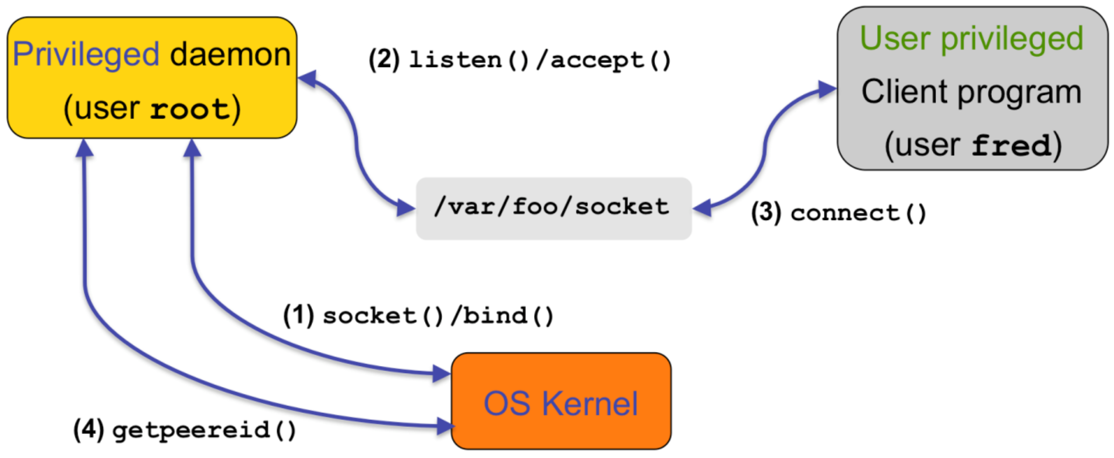

Figure 6: Example Structure of Defer to Kernel Pattern

2.3.5 Participants

These are the participants in the Defer to Kernel pattern:

- Client program. The client program runs with standard user-level privileges. It sends job requests to the server to perform work for which the client lacks sufficient privileges.
- System kernel. The system kernel provides the following:
  - an inter-process communication mechanism used for communication between the client and the server
  - user identity verification and access functionality. Preexisting functionality implemented in the kernel is used to get the ID of a user connected to the server and to check to see if the user's submitted job request is permitted to run on the server.
- Server program. The server program monitors an allocated instance of the IPC mechanism, reads incoming job requests from clients, and checks to see if a client's submitted jobs should be run on the server.

2.3.6 Consequences

Applications that previously relied on a single executable (`setuid` executable on UNIX-based OSs, executable running as administrator under Windows) must be re-architected as a client/server system.

Additional system complexity is added because of the added communication between the client and server.

2.3.7 Implementation

The general implementation of the Defer to Kernel pattern is as follows:

1. The server starts up. It accepts client requests via some known mechanism.
2. The client submits a request to the server. Included with the request is information identifying the client. This information is encoded and/or sent using an existing user identification mechanism inherent to the OS's kernel.
3. The server gets the user request and uses some kernel-level mechanism to determine whether to satisfy the user's request or to reject the request.

A more specific implementation suitable for UNIX-based OSs is as follows:

1. The server (cron job, print job, etc.) opens a UNIX domain socket at a known path. All client requests are directed to this UNIX domain socket.
2. The client connects to this socket to submit a request. Because UNIX domain sockets are being used, information about the client user's UID and GID is automatically included with the message. Note that this is performed by the underlying socket code and does not have to be explicitly programmed into the client.
3. The server, upon receiving the `connect()`, invokes a system call such as `getpeereid()` to identify the user making the request.
4. As with the general pattern, the server uses the user identification information from the previous step to determine whether to satisfy the user's request.

Note that this system only addresses local users and not those connecting remotely over a network.

Linux does not include a `getpeereid()` function. However, `getpeereid()` can easily be implemented as follows:

```
/**
 * Get the user ID and group ID of the user connected to the other end
 * of the given UNIX domain socket.
 *
 * @param sd The UNIX domain socket.
 * @param uid Where to store the user ID of the user connected to the
 * other end of the given UNIX domain socket. Memory for uid must be
 * allocated by the caller.
 * @param gid Where to store the group ID of the user connected to the
 * other end of the given UNIX domain socket. Memory for gid must be
 * allocated by the caller.
 *
 * @returns -1 on failure, 0 on success.
 **/
int getpeereid(int sd, uid_t *uid, gid_t *gid) {
  struct ucred cred;
  int len = sizeof (cred);

  if (getsockopt(sd,SOL_SOCKET,SO_PEERCRED,&cred,&len)) {
    return -1;
  }

  *uid = cred.uid;
  *gid = cred.gid;

  return 0;
}
```

The underlying design of Windows security makes it simple to implement the Defer to Kernel pattern. Under Windows, every process or thread has an associated *access token* containing (among other things) the security identifier (SID) for the user owning the process's account and the SIDs for the user's groups. A server can be set up as a Windows service. A Windows service can be secured by turning it into a *securable object*. When creating a securable object it is possible to associate an *access control list* with the securable object. The access control list contains the SIDs of the client processes that are allowed to connect to the server, that is, the Windows service.

For more information about Windows securable objects, see the online tutorial "Access Control Story: Part I" [Tenouk 2009].

2.3.8 Sample Code

Under Linux, a sketch of the server portion of the Defer to Kernel design pattern is similar to the following:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
```

```
#include <string.h>
#include <unistd.h>
#include <stdio.h>
#include <linux/un.h>

#define SOCKET_ERROR        -1
#define BUFFER_SIZE         100
#define QUEUE_SIZE          5
#define SOCKET_PATH     "/tmp/myserver"

/**
 * Get the user ID and group ID of the user connected to the other end
 * of the given UNIX domain socket.
 *
 * @param sd The UNIX domain socket.
 * @param uid Where to store the user ID of the user connected to the
 * other end of the given UNIX domain socket. Memory for uid must be
 * allocated by the caller.
 * @param gid Where to store the group ID of the user connected to the
 * other end of the given UNIX domain socket. Memory for gid must be
 * allocated by the caller.
 *
 * @returns -1 on failure, 0 on success.
 **/
int getpeereid(int sd, uid_t *uid, gid_t *gid) {
  struct ucred cred;
  socklen_t len = sizeof (cred);

  if (getsockopt(sd,SOL_SOCKET,SO_PEERCRED,&cred,&len)) {
    return -1;
  }

  *uid = cred.uid;
  *gid = cred.gid;

  return 0;
}

/* This refers to a user validation function. This will not be
   implemented in this example.

   The purpose of this function is to check to see if the request of
   the connecting user should be read, that is, it checks to see
   if the connecting user is allowed to submit requests (any request)
   to the server. Note that validateUser() makes use of the user ID and
   the group ID of the user, both of which are gathered using the
   kernel-level getpeereid() function.

   The validity of the actual user request will be checked with the
   validateRequest() function.
*/
extern int validateUser(uid_t uid, gid_t gid);
```

```
/*
   The purpose of this function is to see if the server should honor
   the request of a connected user. Note that as with validateUser(),
   validateRequest() makes use of the user ID and the group ID of the user
   to check to see if the connected user has the rights to make the server
   handle the request.

   This will not be implemented in this example.
*/
extern int validateRequest(uid_t uid, gid_t gid, char *request);

int main(int argc, char* argv[]) {
  int hSocket,hServerSocket;  /* handle to socket */
  struct hostent* pHostInfo;   /* holds info about a machine */
  struct sockaddr_un Address; /* Internet socket address struct */
  int nAddressSize=sizeof(struct sockaddr_in);
  char pBuffer[BUFFER_SIZE];

  /* Make a UNIX domain socket for incoming client requests. */
  hServerSocket=socket(AF_UNIX,SOCK_STREAM,0);

  if (hServerSocket == SOCKET_ERROR) {
    puts("\nCould not make a socket\n");
    return 0;
  }

  /* fill in address structure defining how to set up the
     UNIX domain socket. */
  Address.sun_family = AF_UNIX;
  strcpy(Address.sun_path, SOCKET_PATH);
  unlink(Address.sun_path);

  /* Bind the incoming request socket to a "well-known" path. */
  /* In this simple example the "well-known" path is hard-coded. */
  if(bind(hServerSocket,(struct sockaddr*)&Address,sizeof(Address))
     == SOCKET_ERROR) {
    puts("\nCould not connect to host\n");
    return 0;
  }

  /*  get port number */
  getsockname(hServerSocket,
              (struct sockaddr *) &Address,
              (socklen_t *)&nAddressSize);

  /* Establish the listen queue for the incoming request socket. */
  if (listen(hServerSocket,QUEUE_SIZE) == SOCKET_ERROR) {
    puts("\nCould not listen\n");
    return 0;
  }

  /* Get and handle client requests. */
  for(;;) {
```

```
/* Get a user request via an incoming connection. */
hSocket=accept(hServerSocket,(struct sockaddr*)&Address,
               (socklen_t *)&nAddressSize);

/* Figure out who just connected. */
uid_t connectedUID;
gid_t connectedGID;
if (getpeereid(hSocket, &connectedUID, &connectedGID) != 0) {

  /* We cannot figure out who connected. Boot the connection. */
  puts("Cannot figure out who connected. Booting them.");
  if(close(hSocket) == SOCKET_ERROR) {
    puts("ERROR: Could not close socket\n");
    return 0;
  }

  /* Get more incoming connections. */
  continue;
}

/* Validate the user that is going to make a request. */
if (!validateUser(connectedUID, connectedGID)) {

  puts("User not validated. Booting them.");
  if(close(hSocket) == SOCKET_ERROR) {
    puts("ERROR: Could not close socket\n");
    return 0;
  }

  /* Get more incoming connections. */
  continue;
}

/* Get the user's request. */
char *currRequest;
/* .
   . (The request is pointed to by currRequest.)
   . */

/* Validate the connected user’s request. */
if (!validateRequest(connectedUID, connectedGID, currRequest)) {

  puts("User issued invalid request. Booting them.");
  if(close(hSocket) == SOCKET_ERROR) {
    puts("ERROR: Could not close socket\n");
    return 0;
  }

  /* Get more incoming connections. */
  continue;
}
```

```
    /* Process the user's request. */
    /* .
       .
       . */

    /* Close the socket connected to the current user. */
    if (close(hSocket) == SOCKET_ERROR) {
      puts("ERROR: Could not close socket\n");
      return 0;
    }
  }
```

Under Linux, a sketch of the client portion of the Defer to Kernel design pattern is similar to the following:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <string.h>
#include <unistd.h>
#include <stdio.h>
#include <linux/un.h>
#include <stdlib.h>

#define SOCKET_ERROR        -1
#define BUFFER_SIZE         100
#define SOCKET_PATH     "/tmp/myserver"

int  main(int argc, char* argv[]) {
  int hSocket;                 /* handle to socket */
  struct sockaddr_un Address;  /* internet socket address struct */
  char pBuffer[BUFFER_SIZE];
  unsigned nReadAmount;

  /* Make a UNIX domain socket to use to talk with the server. */
  hSocket=socket(AF_UNIX,SOCK_STREAM,0);
  if (hSocket == SOCKET_ERROR) {
    puts("\nCould not make a socket\n");
    return 0;
  }

  /* fill in address structure defining how to set up the
     UNIX domain socket. */
  Address.sun_family=AF_UNIX;
  strcpy(Address.sun_path, SOCKET_PATH);

  /* Connect to host via a "well known" path. */
  /* In this simple example the "well-known" path is hard-coded. */
  if (connect(hSocket,(struct sockaddr*)&Address,sizeof(Address))
      == SOCKET_ERROR) {
      puts("\nCould not connect to host\n");
      return 0;
  }

  /* Communicate request to the server. */
  /* .
     .
     . */

  /* Close socket used to communicate with the server. */
  if (close(hSocket) == SOCKET_ERROR) {
    puts("\nCould not close socket\n");
    return 0;
  }
}
```

See “Securable Objects” [MSDN 2009a] for information regarding how to implement the Defer to Kernel pattern under Windows.

2.3.9 Known Uses

ucspi-unix

Securable Objects in Windows

3 The Design-Level Patterns

3.1 Secure State Machine

3.1.1 Intent

The intent of the Secure State Machine pattern is to allow a clear separation between security mechanisms and user-level functionality by implementing the security and user-level functionality as two separate state machines.

3.1.2 Also Known As

Secure State

3.1.3 Motivation

Intermixing security functionality and typical user-level functionality in the implementation of a secure system can increase the complexity of both. The increased complexity makes it more difficult to test, review, and verify the security properties of the implementation, increasing the likelihood of introducing a vulnerability.

Also, a tight coupling between the security functionality and the user-level functionality makes it difficult to change and modify the system's security mechanisms.

3.1.4 Applicability

This pattern is applicable if

- the user-level functionality lends itself to implementation using the Gang of Four State pattern [Gamma 1995]; that is, the user-level functionality can be cleanly represented as a finite state machine
- the access control model for the state transition operations in the user-level functionality state machine can also be represented as a state machine. Note that in a degenerate case the access control model could be represented by a state machine with a single state.

3.1.5 Structure

Figure 7 depicts the structure of the Secure State Machine pattern.

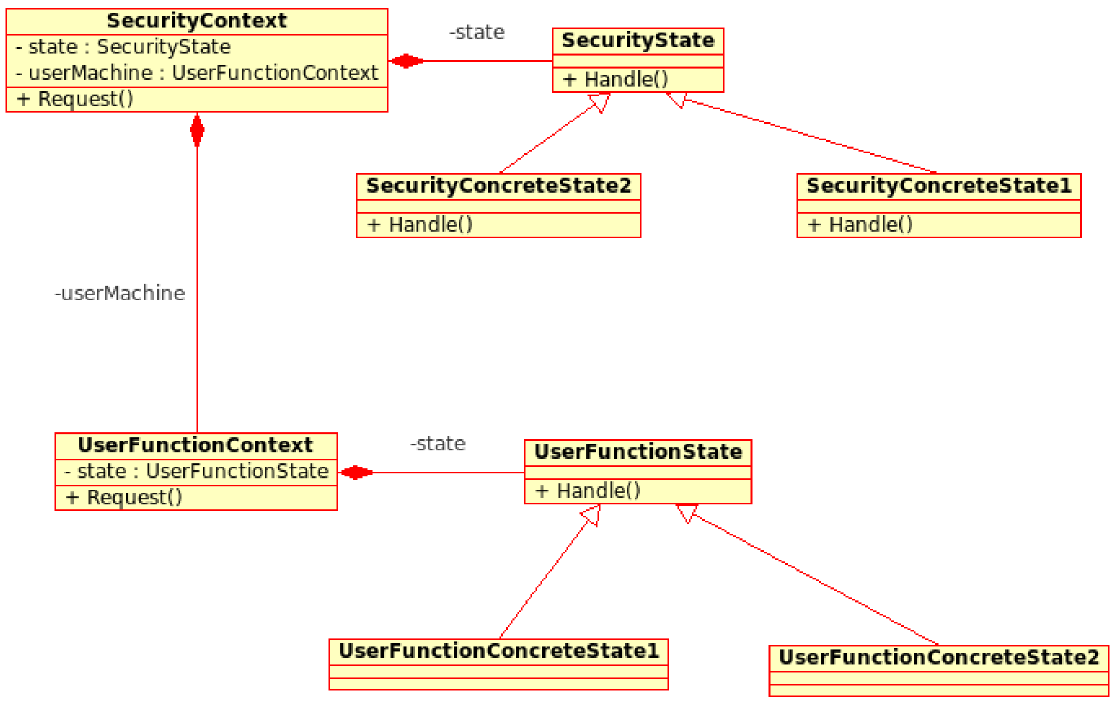

Figure 7: Secure State Machine Pattern Structure

3.1.6 Participants

- **SecurityContext** (ExampleSystem)
  - Defines the interface of interest to clients. All client operations are initially handled by an instance of SecurityContext.
  - As with the original Gang of Four State pattern [Gamma 1995], SecurityContext maintains an instance of a Security-State subclass that defines the current state from a security perspective.
  - Maintains an instance of UserFunctionContext; that is, the state machine implementing the non-security, user-level functionality.
  - Acts as a proxy for the instance of UserFunctionContext.
- **SecurityState** (SecurityState)
  - Defines an interface representing the possible operations handled by the security state machine. Note that UserFunctionState must share the same interface; that is, it must handle the same possible operations as SecureState.
- **SecurityConcreteState** (LoggedOut, LoggedInAdmin, LoggedInClerk, Locked)
  - Each subclass of SecurityState implements the security state-dependent behavior for each operation.

The components of the user-level functionality state machine are exactly the same as those in the Gang of Four State pattern.

- **UserFunctionContext** (UserFunctionsMachine)

  - Defines all of the same operations as SecurityContext so that components of the security state machine can forward operation requests to the user-level functionality state machine when appropriate.
  - Has a private constructor to prevent outside access to the functionality of the user-level state machine. Only a SecurityContext can create a new UserFunctionContext.
- **UserFunctionState** (UserFunctionState)
  - Has the same interface as SecurityState.
- **UserFunctionConcreteState** (UserFunctionConcreteState1, UserFunctionConcreteState2)
  - In a manner similar to SecurityConcreteState, each subclass of SecurityState implements the user-level state-dependent behavior for each operation.

3.1.7 Consequences

In addition to the set of consequences associated with the general State pattern, the Secure State Machine pattern has these additional consequences:

- *It clearly separates security mechanisms from user-level functionality*. The use of this pattern requires that the security mechanisms be explicitly implemented in the security state machine and the user functionality of the system be explicitly implemented in the user-level functionality state machine. This makes it easy to
  - test and verify the security mechanisms separately from the user-level functionality. Because the security functionality is implemented separately from the user-level functionality, more rigorous testing and verification techniques can be applied to the security state machine than to the user-level functionality state machine.
  - change or replace the security mechanism. Because the security functionality is separate from the user-level functionality, a new security implementation could be implemented with less effort than would be required if the existing security mechanisms were interleaved with the user-level functionality.
- *It prevents programmatic access to the user-level functionality that avoids security*. Because only the security state machine can create an instance of the user-level functionality state machine, all interaction with the user-level functionality state machine must first pass through the security state machine, consequently defeating one class of programmatic attack.

3.1.8 Implementation

In addition to the implementation considerations associated with the Gang of Four State pattern [Gamma 1995], the Secure State Machine pattern has the following implementation consideration.

Who forwards operations on to the user-level state machine? The operations handled by the security state machine can be forwarded on to the user-level state machine by either the SecurityContext instance or the SecurityConcreteState instance.

- SecurityContext instance. The forwarding of operations to the user-level functionality state machine can be handled in the SecurityContext instance by defining the operation methods in SecurityState to return a boolean value indicating whether the operation should be forwarded. The corresponding operation methods in SecurityContext would then use this return value to determine whether to forward the operation. This method is used in the example on this page. It is recommended over performing the forwarding in the SecurityConcreteState instance because it allows the user functionality state machine to be completely hidden within the SecurityContext instance.

- SecurityConcreteState instance. If the SecurityContext provides a method by which a SecurityConcreteState can access the user-level functionality state machine, the SecurityConcreteState can forward the operation directly.

3.1.9 Sample Code

This example of using the Secure State Machine pattern provides a skeleton of the code for implementing a system with the following behavior:

- A user must log in before using the system.
- If there are five failed login attempts, the user's account will be locked.
- Each user will be handled by a separate state machine. The allocation of users to state machines will be handled by some other portion of the system.
- The user-level functionality is abstractly represented as `op1`, `op2`, `op3`, `login`, and `logout`.
- For security reasons, `op3` may be performed only 50 times in a session. If `op3` is performed more than 50 times, the user will be automatically logged out.
- Performing `op2` requires that the user have the role of administrator. Everyone else has the role of clerk.

A collaboration diagram describing the basic behavior of the example code is shown in Figure 8.

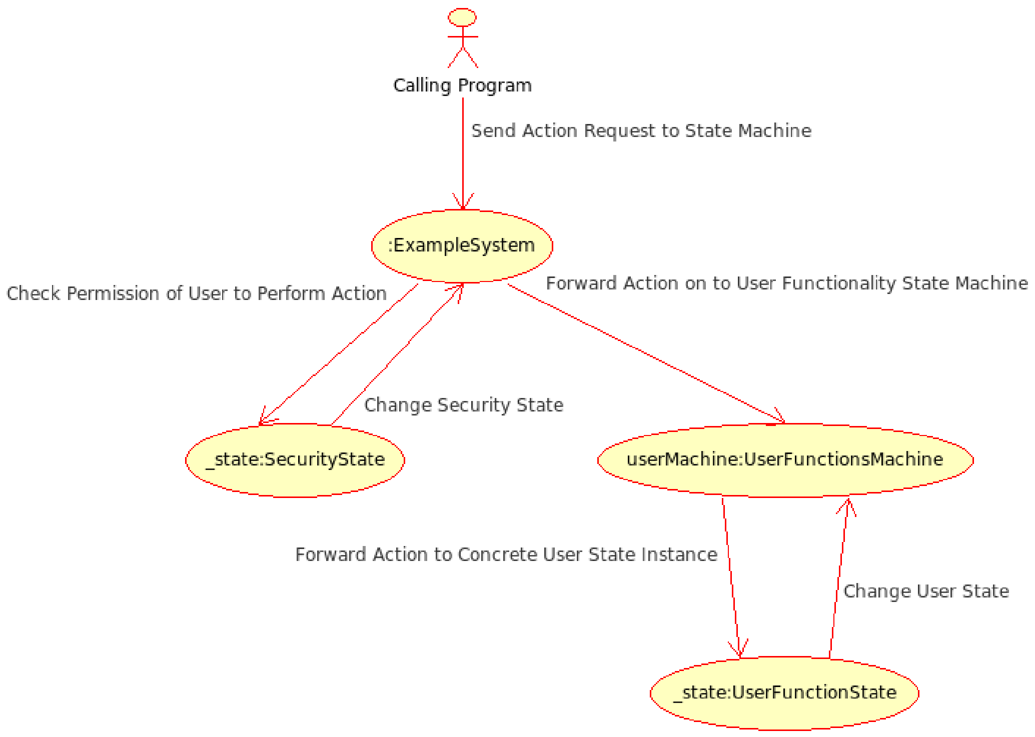
Calling Program
Send Action Request to State Machine
:ExampleSystem
Check Permission of User to Perform Action
Forward Action on to User Functionality State Machine
Change Security State
_state:SecurityState
userMachine:UserFunctionsMachine
Forward Action to Concrete User State Instance
Change User State
_state:UserFunctionState

Figure 8: Secure State Machine Example Code Collaboration Diagram

This class represents the credentials of the user associated with a state machine. This class will not be sketched out in this example.

```
class UserCredentials;
```

This class represents a string that has been encrypted. This class will not be sketched out in this example.

```
class EncryptedString;
```

This is the forward declaration abstract class representing the states of the security state machine. This class will be sketched out in the example.

```
class SecurityState;
```

This class implements the security state machine for the system and acts as a proxy for the state machine that actually implements the user-level functionality.

```
class ExampleSystem {

public:

  // Create a new example system state machine for the given user.
  ExampleSystem(UserCredentials user);

  // Someone is trying to log onto the system as the current user.
  void login(EncryptedString password);

  // The user is logging out.
  void logout();

  // The user is attempting to perform one of the three user-level
  // system operations.
  void op1();
  void op2();
  void op3();

private:

  // Track the current state of the security state machine.
  SecurityState* _state;

  // Change the current state in the controller.
  void changeState(SecurityState*);

  // Let the security state machine change the current state in the
  // controller.
  friend class SecurityState;

  // Track the user associated with the security state machine.
  UserCredentials _user;

  // Get the user associated with the security state machine.
  const UserCredentials getUser();
  // Track the state machine that actually implements the user
  // functionality. The security state machine acts as a proxy for
  // the user functionality state machine.
  UserFunctionsMachine userMachine;
};
```

The methods of the SecurityContext are defined as follows.

```
ExampleSystem::ExampleSystem(UserCredentials user) {

  // Initially the user is logged out.
  _state = LoggedOut::instance(user);

  // We need to save the user we are dealing with.
  _user = user;

  // Create the user-level state machine for which we are a proxy.
  userSystem = UserFunctionsMachine(user);
}

void ExampleSystem::login(EncryptedString password) {
  // Forward the operation if appropriate.
  if (_state->login(this, password)) {
    userMachine->login(password);
  }
}

void ExampleSystem::logout() {
  // Forward the operation if appropriate.
  if (_state->logout(this)) {
    userMachine->logout();
  }
}

void ExampleSystem::op1() {
  // Forward the operation if appropriate.
  if (_state->op1(this)) {
    userMachine->op1();
  }
}

void ExampleSystem::op2() {
  // Forward the operation if appropriate.
  if (_state->op2(this)) {
    userMachine->op2();
  }
}

void ExampleSystem::op3() {
  // Forward the operation if appropriate.
  if (_state->op3(this)) {
    userMachine->op3();
  }
}
```

This is the declaration of the abstract class defining the interface for the state classes defining the states of the security state machine.

```
class SecurityState {

public:

  virtual bool login(ExampleSystem* controller, EncryptedString password);
  virtual bool logout(ExampleSystem* controller);
  virtual bool op1(ExampleSystem* controller);
  virtual bool op2(ExampleSystem* controller);
  virtual bool op3(ExampleSystem* controller);

protected:

  void changeState(ExampleSystem* controller, SecurityState* newState);
};
```

The security model for this example has four states:

- NotLoggedIn. The user is not logged in.
- LoggedInAdmin. The user is logged in as an administrator.
- LoggedInClerk. The user is logged in as a clerk.
- Locked. The user's account has been locked.

The default implementation of the security state methods is as follows. These statements should be redefined by the concrete state classes. In the default implementation, the operation is never forwarded on to the user-level functionality state machine.

```
bool SecurityState::login(ExampleSystem* controller,
                          EncryptedString password) { return false; }
bool SecurityState::logout(ExampleSystem* controller) { return false; }
bool SecurityState::op1(ExampleSystem* controller) { return false; }
bool SecurityState::op2(ExampleSystem* controller) { return false; }
bool SecurityState::op3(ExampleSystem* controller) { return false; }
```

`changeState()` is common to all concrete state classes.

```
void SecurityState::changeState(ExampleSystem* controller,
                                SecurityState* newState) {
  controller->changeState(newState);
}
```

Here is the definition of the concrete `NotLoggedIn` state class.

```
class NotLoggedIn : public SecurityState {

public:

  // Get an instance of this state for the current user. Each user
  // will have a single instance of each security state associated
  // with them. This ensures that each user will be associated with
  // one and only one security state machine.
  static SecurityState* instance(UserCredentials user);

  // When the user is not logged in, all they can do is try to log
  // in.
  virtual bool login(ExampleSystem* controller, EncryptedString password);

private:

  // This state will track the number of failed login attempts.
  unsigned int numFailedLogins;
};
```

Here are the method bodies of the `NotLoggedIn` state class.

Create a `NotLoggedIn` state. This initializes the number of failed login attempts.

```
NotLoggedIn::NotLoggedIn() {
  numFailedLogins = 0;
}
```

Handle a user login.

```
bool NotLoggedIn::login(ExampleSystem* controller, EncryptedString password)
{

  // Try to validate the user with the password.
  if (controller->getUser().validate(password)) {

    // The current user correctly entered their password.

    // Clear the bad password count.
    numFailedLogins = 0;

    // They user is now logged in. Choose the proper login state based
    // on the user's role.
    if (controller->getUser().isAdministrator()) {
      changeState(controller, LoggedInAdmin::instance());
    }
    else {
      changeState(controller, LoggedInClerk::instance());
    }

    // The user has now logged in. Handle the user functionality
    // associated with a login by passing the login operation on to
    // the user functionality machine.
    return true;
  }

  else {
    // The current user incorrectly entered their password.

    // Track the failed login.
    numFailedLogins++;
    // Has the user failed their login too many times.
    if (numFailedLogins >= 5) {

      // Reset the # of failed logins.
      numFailedLogins = 0;

      // Lock the user’s account.
      changeState(controller, Locked::instance());

      // Note that because the security state machine determined that
      // the security requirements were not met, the login operation
      // is not passed on to the user functionality machine.
      return false;
    }
  }
}
```

Here is the definition of the concrete `Locked` state class.

```
class Locked : public SecurityState {

public:

  static SecurityState* instance(UserCredentials user);

  // For this simple example, once a user's account is locked it
  // cannot be unlocked. Once the user's account is locked, they
  // cannot do anything. No operations are forwarded to the user
  // functionality machine.
};
```

Here is the definition of the concrete `LoggedInAdmin` state class

```
class LoggedInAdmin : public SecurityState {

public:

  static SecurityState* instance(UserCredentials user);
  LoggedInAdmin();

  // A logged-in administrator can perform all operations other than
  // logging in again.
  bool logout(ExampleSystem* controller);
  bool op1(ExampleSystem* controller);
  bool op2(ExampleSystem* controller);
  bool op3(ExampleSystem* controller);

private:

  // Keep track of the number of times the user has performed
  // op3.
  unsigned int op3Count;
};
```

Here are the method bodies of the `LoggedInAdmin` state class.

```
// Create a LoggedInAdmin state. This initializes the count of the
// number of times op3 was performed.
LoggedInAdmin::LoggedInAdmin() {
  op3Count = 0;
}

bool LoggedInAdmin::logout(ExampleSystem* controller) {

  // Just move to the logged out state.
  changeState(controller, LoggedOut::instance());

  // Handle user functionality actions for the logout operation.
  return true;
}

bool LoggedInAdmin::op1(ExampleSystem* controller) {
  // Based on the current state of the security machine we know that
  // this operation is valid. Forward it on to the user functionality
  // machine.
  return true;
}

bool LoggedInAdmin::op2(ExampleSystem* controller) {
  // Based on the current state of the security machine we know that
  // this operation is valid. Forward it on to the user functionality
  // machine.
  return true;
}

bool LoggedInAdmin::op3(ExampleSystem* controller) {

  // The user has done op3 one more time. Track it.
  op3Count++;

  // Has the user exceeded their quota of # of times they can do
  // op3?
  if (op3Count > 50) {

    // Reset the count of # of times they performed op3 during this
    // login session.
    op3Count = 0;

    // Log out the user. Note that this calls the controller’s logout
    // method, which will result in both the security machine and the
    // user-level functionality machine handling the logout
    // operation.
    controller->logout();

    // Stop processing the op3 operation.
    return false;
  }
  // If we get here the security criteria for op3 have been
  // met. Forward op3 on to the user functionality machine.
  return true;
}
```

Here is the definition of the concrete `LoggedInClerk` state class.

```
class LoggedInClerk : public SecurityState {

public:

  static SecurityState* instance(UserCredentials user);
  LoggedInClerk();

  // A logged in clerk can perform all operations other than
  // logging in again and op2.
  bool logout(ExampleSystem* controller);
  bool op1(ExampleSystem* controller);
  bool op3(ExampleSystem* controller);

private:

  // Keep track of the number of times the user has performed
  // op3.
  unsigned int op3Count;
};
```

Here are the method bodies of the `LoggedInClerk` state class.

```
// Create a LoggedInClerk state. This initializes the count of the
// number of times op3 was performed.
LoggedInClerk::LoggedInClerk() {
  op3Count = 0;
}

bool LoggedInClerk::logout(ExampleSystem* controller) {

  // Just move to the logged out state.
  changeState(controller, LoggedOut::instance());

  // Handle user functionality actions for the logout operation.
  return true;
}

bool LoggedInClerk::op1(ExampleSystem* controller) {
  // Based on the current state of the security machine we know that
  // this operation is valid. Forward it on to the user functionality
  // machine.
  return true;
}

bool LoggedInClerk::op3(ExampleSystem* controller) {

  // The user has done op3 one more time. Track it.
  op3Count++;

  // Has the user exceeded their quota of # of times they can do
  // op3?
  if (op3Count > 50) {

    // Reset the count of # of times they performed op3 during this
    // login session.
    op3Count = 0;

    // Log out the user. Note that this calls the controller’s logout
    // method, which will result in both the security machine and the
    // user-level functionality machine handling the logout
    // operation.
    controller->logout();

    // Stop processing the op3 operation.
    return false;
  }

  // If we get here the security criteria for op3 have been
  // met. Forward op3 on to the user functionality machine.
  return true;
}
```

This is the controller for the user-level functionality state machine. Note that only the security state machine can create an instance of the user-level functionality state machine.

```
class UserFunctionsMachine {

public:

  // Someone is trying to log onto the system as the current user.
  void login(EncryptedString password);

  // The user is logging out.
  void logout();

  // The user is attempting to perform one of the three user-level
  // system operations.
  void op1();
  void op2();
  void op3();

private:

  // Only the security state machine can create an instance of the
  // user-level functionality machine. This helps prevent direct
  // access to the user-level functionality machine.
  friend class ExampleSystem;

  // Create a new user functionality state machine for the given user.
  UserFunctionsMachine(UserCredentials user);
};
```

3.1.10 Known Uses

"Method and apparatus for secure context switching in a system including a processor and cached virtual memory" (United States Patent Application 20070260838).

3.2 Secure Visitor

3.2.1 Intent

Secure systems may need to perform various operations on hierarchically structured data where each node in the data hierarchy may have different access restrictions; that is, access to data in different nodes may be dependent on the role/credentials of the user accessing the data. The Secure Visitor pattern allows nodes to *lock* themselves against being read by a visitor unless the visitor supplies the proper credentials to *unlock* the node. The Secure Visitor is defined so that the only way to access a locked node is with a visitor, helping to prevent unauthorized access to nodes in the data structure.

3.2.2 Motivation

As with the Secure State Machine pattern, the primary motivation of the Secure Visitor pattern is to provide a clean separation between security considerations and user-level functionality. The Secure Visitor pattern allocates all of the security considerations to the nodes in the data hierarchy, leaving developers free to write visitors that only concern themselves with user-level functionality.

Making the nodes in the data hierarchy solely responsible for security functionality makes it more feasible to test and verify the security functionality more rigorously than the user-level functionality. It also frees the user functionality developers from having to reimplement security functionality each time a new visitor is developed, thereby avoiding the creation of new security holes.

3.2.3 Applicability

This pattern is applicable if

- the system possesses hierarchical data that can be processed using the original Gang of Four Visitor pattern [Gamma 1995]
- various nodes in the hierarchical data have different access privileges

3.2.4 Structure

Figure 9 shows the structure of the Secure Visitor pattern.

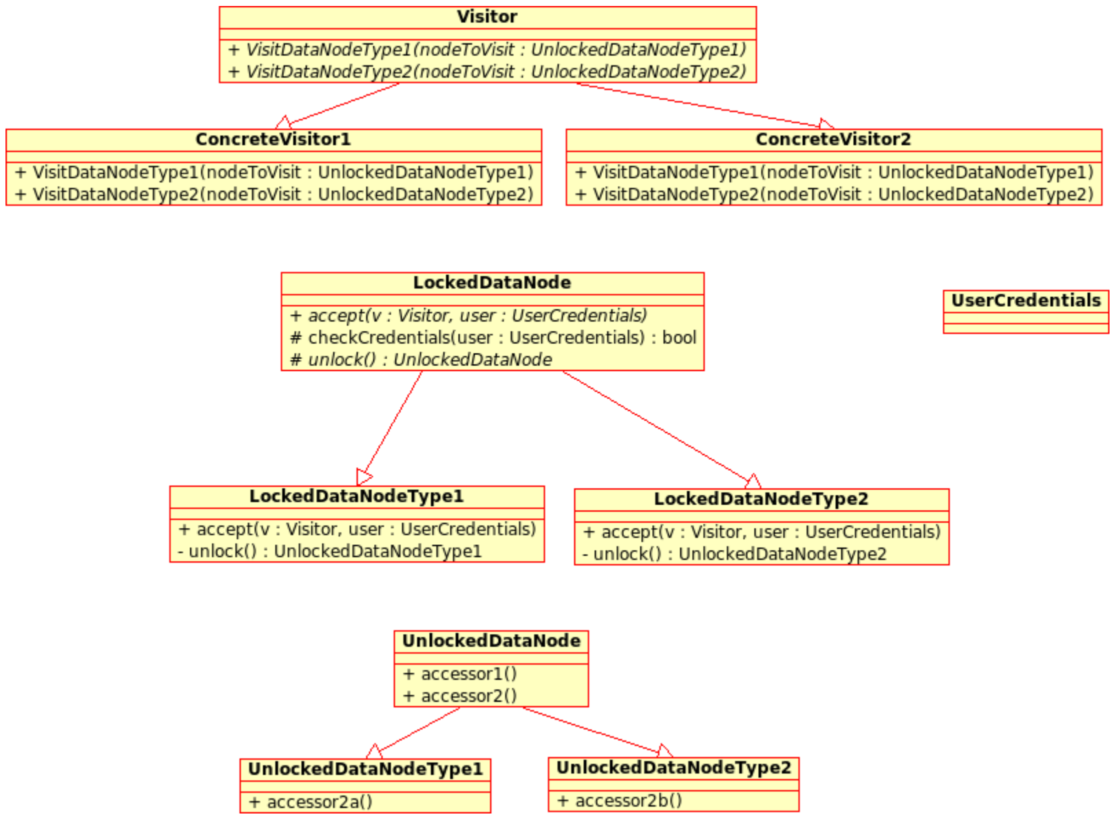
Visitor
+ VisitDataNodeType1(nodeToVisit : UnlockedDataNodeType1)
+ VisitDataNodeType2(nodeToVisit : UnlockedDataNodeType2)
ConcreteVisitor1
+ VisitDataNodeType1(nodeToVisit : UnlockedDataNodeType1)
+ VisitDataNodeType2(nodeToVisit : UnlockedDataNodeType2)
ConcreteVisitor2
+ VisitDataNodeType1(nodeToVisit : UnlockedDataNodeType1)
+ VisitDataNodeType2(nodeToVisit : UnlockedDataNodeType2)
LockedDataNode
+ accept(v : Visitor, user : UserCredentials)
# checkCredentials(user : UserCredentials) : bool
# unlock() : UnlockedDataNode
UserCredentials
LockedDataNodeType1
+ accept(v : Visitor, user : UserCredentials)
- unlock() : UnlockedDataNodeType1
LockedDataNodeType2
+ accept(v : Visitor, user : UserCredentials)
- unlock() : UnlockedDataNodeType2
UnlockedDataNode
+ accessor1()
+ accessor2()
UnlockedDataNodeType1
+ accessor2a()
UnlockedDataNodeType2
+ accessor2b()

Figure 9: Secure Visitor Pattern Structure

3.2.5 Participants

These are the participants in the Secure Visitor pattern. (The class in the code presented in the Sample Code section corresponding to the listed participant appears in parentheses after the participant.)

- **Visitor** (HierarchicalDataVisitor). The Visitor participant in the secure visitor pattern is almost exactly the same as the Visitor participant in the standard Visitor pattern. The primary difference in the patterns is that the various visit methods take unlocked node objects in the Secure Visitor pattern, whereas the visit methods in the standard Visitor pattern simply take a node object (the standard Visitor pattern has no concept of locked and unlocked data nodes).
- **ConcreteVisitor**. As with the standard Visitor pattern, the ConcreteVisitor classes implement the operations defined in the abstract Visitor class.
- **LockedDataNode** (LockedDataNode). The LockedDataNode class defines an `accept()` operation that accepts a visitor. In addition, the LockedDataNode class also defines an operation for checking a user's credentials and for unlocking the current locked node. Note that a locked node presents no public operations for viewing the data in the node or changing the data in the node. All access to the node must be directed through the node's `accept()` operation. The `accept()` operation will check the user's credentials. If the credentials are valid for the user to view the data in the current node, the node will unlock itself using the `unlock()` operation and pass the unlocked version of itself to the visit method of the visitor.

- **LockedDataNodeTypeN** (LockedDataNodeType1). The LockedDataNodeTypeN classes implement the operations defined in the abstract LockedDataNode class. This includes the `unlock()` operation to unlock the various locked node objects and return the unlocked versions of the nodes.
- **UnlockedDataNode** (UnlockedDataNode). This class represents the unlocked version of a locked data node. The unlocked version of a node has some important characteristics:
  - It has no access to the parent(s) or children of its corresponding locked node. It only contains the data specific to the node itself, that is, the data that the user has been granted permission to see.
  - It has no `accept()` operation. The traversal of a hierarchical data structure with a secure visitor is done on the locked nodes, not the unlocked nodes.
- **UnlockedDataNodeTypeN** (UnlockedNodeType1). The concrete implementations of UnlockedDataNode implement the operations defined in the abstract class.
- **UserCredentials**. The UserCredentials represent the current user of the system and/or the permissions assigned to the current user. The Secure Visitor pattern does not place many restrictions on the specific implementation of the user credentials. The only requirement is that it is possible for a node to use the credentials to control access to the node's data.

3.2.6 Consequences

In addition to the set of consequences associated with the standard Visitor pattern, the Secure Visitor pattern has these additional consequences:

- *It clearly separates security mechanisms from user-level functionality*. The use of this pattern requires that the nodes in the data hierarchy, not the visitors themselves, implement security. This makes it easy to
  - test and verify the security aspects separately from the user-level functionality. Because the security functionality is implemented separately from the user-level functionality, more rigorous testing and verification techniques can be applied to the security state machine than to the user-level functionality state machine.
  - change or replace the security mechanism. Because the security functionality is implemented in the nodes in the data hierarchy and not in the various visitors of the data hierarchy, the security mechanism can be changed without requiring any modifications to the visitors.
- *It prevents programmatic access to the user-level functionality that avoids security*. Because the only way to access a locked node in the data hierarchy is via the `accept()` method of the Visitor pattern and the only class allowed to create an unlocked version of a node is its corresponding locked node, it is difficult or impossible to programmatically access the data in a node without supplying valid credentials for the node.

3.2.7 Implementation

In addition to the implementation considerations associated with the standard Visitor pattern, the Secure Visitor pattern has the following implementation consideration:

How is the data in a locked node protected? The goal of the Secure Visitor design pattern is to make it difficult to read the data in a locked node without supplying the appropriate credentials for the node. While the pattern itself makes it difficult to programmatically read a locked node data without the appropriate credentials, it still may be possible to read the raw bytes making up the locked node and thereby gain access to the data in the locked node. This implies that the data in the locked node must actually be “locked” in some manner. Data can be locked in a locked node using encryption or off-line storage.

- **Encryption**. The data in a locked node can be encrypted and only decrypted as part of the process of making an unlocked version of the node after accepting the credentials of a visitor.
- **Off-line storage**. The actual data in a locked node can be stored in some sort of an external, protected data management system like a database. The actual node data would only be loaded from the external source after accepting the credentials of a visitor.

3.2.8 Sample Code

The following collaboration diagram represents the basic behavior of the example code presented in this section:

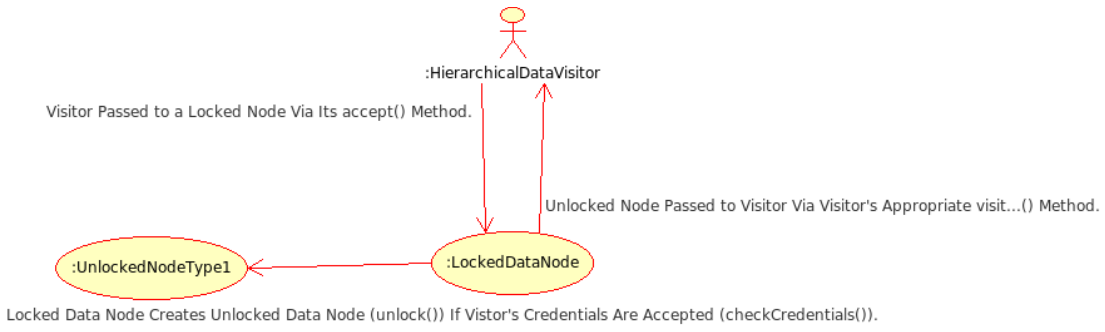
:HierarchicalDataVisitor
Visitor Passed to a Locked Node Via Its accept() Method.
Unlocked Node Passed to Visitor Via Visitor's Appropriate visit...() Method.
:UnlockedNodeType1
:LockedDataNode
Locked Data Node Creates Unlocked Data Node (unlock()) If Vistor's Credentials Are Accepted (checkCredentials()).

Figure 10: Secure Visitor Example Code Collaboration Diagram

This class represents the credentials of the user associated with a visitor; that is, the visitor is visiting the data in response to some action performed by the user represented by the given credentials. This class will not be sketched out in this example.

```
class UserCredentials;
```

This is a forward declaration for an unlocked version of the locked node. This will be defined later in the example.

```
class UnlockedDataNode;
```

This is a forward declaration for the visitor of the locked nodes in the hierarchical data. This will be defined later in the example.

```
class HierarchicalDataVisitor;
```

This defines the general interface for a locked node in the Secure Visitor pattern. It looks just like the interface in the standard Visitor pattern.

```
class LockedDataNode {
```

```
public:
  virtual void accept(HierarchicalDataVisitor& visitor,
                      UserCredentials user);

private:

  // Each type of node will have some way of checking the visitor’s
  // credentials to see if the user has permission to access the
  // node.
  virtual bool checkCredentials(UserCredentials user);
};
```

The visitor interface in the Secure Visitor looks very much like the visitor interface in the standard Visitor pattern. The only difference is that the various `visit…()` methods accept the unlocked version of a node, not the locked version.

```
class HierarchicalDataVisitor {

public:

  virtual ~HierarchicalDataVisitor()
  virtual void visitNodeType1(UnlockedNodeType1 *node);

protected:

  HierarchicalDataVisitor();
};
```

Each concrete node in the data hierarchy has both a locked and unlocked version. Only a locked node will be able to create an unlocked node.

```
class LockedNodeType1 : LockedDataNode {

public:

  // The only way to access the data in the data hierarchy in the
  // Secure Visitor pattern is via the accept() method that accepts a
  // node visitor and the current user's credentials.
  void accept(HierarchicalDataVisitor& visitor,
              UserCredentials user);

private:

  // If the locked node accepts the visitor's credentials, it will
  // create an unlocked version of itself to pass to the visitor for
  // processing. Only a LockedDataNode can create an
  // UnlockedDataNode.
  UnlockedNodeType1 unlock();

  // Each type of node will have some way of checking the visitor’s
  // credentials to see if the user has permission to access the
  // node.
  bool checkCredentials(UserCredentials user);
  // Track the children of the node somehow...
  // ...
};
```

The accept method for a locked node in the data hierarchy checks the user's credentials and unlocks the node and passes it on to the visitor if the credentials are valid for the node.

```
void LockedNodeType1::accept(HierarchicalDataVisitor& visitor,
                             UserCredentials user) {

  // Are the credentials valid for this node?
  if (checkCredentials(user)) {

    // The user has access to this node. Unlock the node and pass it
    // on to the visitor.
    visitor.visitNodeType1(unlock());
  }

  // Visit the children of the node...
  // ...
}
```

Note that the constructor for an unlocked node is private and that the corresponding locked node class is its friend. This means that an unlocked node can be created only by a locked node.

```
class UnlockedNodeType1 {

public:

  ...Data access methods, etc. ...

private:
  UnlockedNodeType1();
  friend class LockedNodeType1;
}
```

4 The Implementation-Level Patterns

4.1 Secure Directory

4.1.1 Intent

The intent of the Secure Directory pattern is to ensure that an attacker cannot manipulate the files used by a program during the execution of the program. See “FIO15-C. Ensure that file operations are performed in a secure directory” in *The CERT C Secure Coding Standard* [Seacord 2008] for additional information regarding this issue.

4.1.2 Motivation

A program may depend on a file for some length of time during program execution. The program developers usually assume that the files used by the program will not be manipulated by outside users during the execution of the program. However, if this assumption is false, a file may be modified by multiple users, which means that a malicious user may modify or delete the file during a critical time when the program relies on the file remaining unmodified, causing a race condition in the program.

The Secure Directory pattern ensures that the directories in which the files used by the program are stored can only be written (and possibly read) by the user of the program.

4.1.3 Applicability

The Secure Directory pattern is applicable for use in a program if

- the program will be run in an insecure environment; that is, an environment where malicious users could gain access to the file system used by the program
- the program reads and/or writes files
- program execution could be negatively affected if the files read or written by the program were modified by an outside user while the program was running

4.1.4 Structure

Programmatically, the structure of the Secure Directory pattern is fairly simple. Prior to opening a file for reading or writing, the Secure Directory pattern states that the program must

1. find the canonical pathname of the directory of the file (see Section 4.2, “Pathname Canonicalization”)
2. check to see if the directory, as referenced by the canonical pathname, is secure

The structure of the secure directory is such that the directory has write permissions limited to the user and the superuser. No other users may modify files in the secure directory. Furthermore, all directories that appear before the directory of interest must prevent other users from renaming or deleting the secure directory.

4.1.5 Participants

The participants in the Secure Directory pattern are

- the program reading and writing the file
- the file system

4.1.6 Consequences

Secure Directory reduces the possibility of race conditions occurring between programs controlled by different users. Race conditions involving a secure directory may be produced only by multiple programs under the control of the user.

The program speed will be degraded due to the canonicalization of pathnames and the checking for secure directories. To reduce the overhead of checking for secure directories, it is possible to cache the result of checking the security of a particular directory. Note that the caching of secure directory results assumes that the permissions of directories used by the program are not changed during program execution.

4.1.7 Implementation

Unless a program is run with root privileges, it does not have the ability to create secure directories. Therefore, the program should check that a directory offered to it is secure, and refuse to use it otherwise. As discussed in the Structure section, the basic implementation of the Secure Directory pattern involves the following steps:

1. Find the canonical pathname of the directory of the file to be read or written. (See Section 4.2, “Pathname Canonicalization.”)
2. Check to see if the directory, as referenced by the canonical pathname, is secure.
   - If the directory is secure, read or write the file.
   - If the directory is not secure, issue an error and do not read or write the file.

4.1.8 Sample Code

The sample code provided in this section was taken directly from “FIO15-C. Ensure that file operations are performed in a secure directory” in *The CERT C Secure Coding Standard* [Seacord 2008].

Under a POSIX-compliant OS, a function to check a directory to see if it is secure may be implemented as follows:

```
#include <stdlib.h>
#include <unistd.h>
#include <limits.h>
#include <libgen.h>
#include <sys/stat.h>
#include <string.h>

/* Returns nonzero if directory is secure, zero otherwise */
int secure_dir(const char *fullpath) {
  char *path_copy = NULL;
  char *dirname_res = NULL;
  char **dirs = NULL;
  int num_of_dirs = 0;
  int secure = 1;
  int i;
  struct stat buf;
  uid_t my_uid = geteuid();

  if (!(path_copy = strdup(fullpath))) {
    /* Handle error */
  }

  dirname_res = path_copy;
  /* Figure out how far it is to the root */
  while (1) {
    dirname_res = dirname(dirname_res);

    num_of_dirs++;

    if ((strcmp(dirname_res, "/") == 0) ||
   (strcmp(dirname_res, "//") == 0)) {
      break;
    }
  }
  free(path_copy);
  path_copy = NULL;

  /* Now allocate and fill the dirs array */
  if (!(dirs = (char **)malloc(num_of_dirs*sizeof(*dirs)))) {
    /* Handle error */
  }
  if (!(dirs[num_of_dirs - 1] = strdup(fullpath))) {
    /* Handle error */
  }

  if (!(path_copy = strdup(fullpath))) {
    /* Handle error */
  }

  dirname_res = path_copy;
  for (i = 1; i < num_of_dirs; i++) {
    dirname_res = dirname(dirname_res);

    dirs[num_of_dirs - i - 1] = strdup(dirname_res);

  }
  free(path_copy);
  path_copy = NULL;

  /* Traverse from the root to the leaf, checking
   * permissions along the way */
  for (i = 0; i < num_of_dirs; i++) {
    if (stat(dirs[i], &buf) != 0) {
       /* Handle error */
    }
    if ((buf.st_uid != my_uid) && (buf.st_uid != 0)) {
      /* Directory is owned by someone besides user or root */
      secure = 0;
    } else if ((buf.st_mode & (S_IWGRP | S_IWOTH))
      && ((i == num_of_dirs - 1) || !(buf.st_mode & S_ISVTX))) {
        /* Others have permissions to the leaf directory
         * or are able to delete or rename files along the way */
        secure = 0;
     }

    free(dirs[i]);
    dirs[i] = NULL;
  }

  free(dirs);
  dirs = NULL;

  return secure;
}
```

Given the `secure_dir()` function, the Secure Directory pattern may be implemented in C as follows:

```
char *dir_name;
char *canonical_dir_name;
const char *file_name = "passwd"; /* filename within the secure directory */
FILE *fp;

/* initialize dir_name */

canonical_dir_name = realpath(dir_name, NULL);
if (canonical_dir_name == NULL) {
  /* Handle error */
}

if (!secure_dir(canonical_dir_name)) {
  /* Handle error */
}

if (chdir(canonical_dir_name) == -1) {
  /* Handle error */
}

fp = fopen(file_name, "w");
if (fp == NULL) {
  /* Handle error */
}

/*... Process file ...*/

if (fclose(fp) != 0) {
  /* Handle error */
}
```

```
if (remove(file_name) != 0) {
  /* Handle error */
}
```

4.2 Pathname Canonicalization

4.2.1 Intent

The intent of the Pathname Canonicalization pattern is to ensure that all files read or written by a program are referred to by a valid path that does not contain any symbolic links or shortcuts, that is, a canonical path.

4.2.2 Motivation

Because of symbolic links and other file system features, a file may not actually reside in the directory indicated by a path. Therefore, performing string-based validation on the pathname may yield false results. Having the true, canonical pathname is particularly important when checking a directory to see if it is secure.

4.2.3 Applicability

The use of the Pathname Canonicalization pattern is applicable if all of the following conditions are true:

- the program accepts pathnames from untrusted sources
- an attacker could provide a pathname to the system that non-obviously refers to a directory or file to which the attacker should not have access
- the program runs in an environment where each file has a unique canonical pathname

4.2.4 Structure

Programmatically, the structure of the Pathname Canonicalization pattern involves calling an OS-specific pathname canonicalization function on the given pathname prior to opening the file. The canonicalized pathname is used when operating on the file.

The canonicalized pathname itself has a structure such that every element of the canonicalized path, except the last, is the genuine directory, and not a link or shortcut. The last element is the genuine filename, and not a link or shortcut.

4.2.5 Participants

The participants in the Pathname Canonicalization pattern are

- the program opening file(s)
- the file system (potentially, depending on the implementation of the OS-specific canonicalization function)

4.2.6 Consequences

Pathname canonicalization guarantees that textual analysis of the canonicalized pathname yields accurate results, which improves the accuracy and security of file access.

The program speed is degraded due to the canonicalization of pathnames. To reduce the overhead of canonicalization, it is possible to cache the canonicalized pathname. Note that such caching assumes that the directory structure accessed by the program is not changed during program execution.

4.2.7 Implementation

The core of the implementation of this pattern is an OS-specific function for performing pathname canonicalization. The canonicalization function is a routine that would ensure that every directory in a pathname is a genuine directory rather than a link or shortcut. The result of the canonicalization function is a canonicalized path such that string-based validation of the path always yields valid results. For instance, a canonicalized path that begins with the pathname to a user's home directory will guarantee that the path's file lives in the user's home directory or a subdirectory below the user's home directory.

As discussed in the Structure section, given the canonicalization function, the implementation of the Pathname Canonicalization pattern is fairly simple:

1. The program calls the OS-specific pathname canonicalization function on the given pathname prior to opening a file.
2. The canonicalized pathname is used when operating on the file.

Canonicalization routines should be provided by the platform; a program should simply call the platform's canonicalization routine before performing textual analysis on a pathname. Some OS-specific canonicalization functions are

- POSIX-compliant OSs: `realpath()`
- systems with `glibc`: `canonicalize_file_name()`, a GNU extension provided in `glibc`

See FIO02-C, "Canonicalize path names originating from untrusted sources for implementation details" in *The CERT C Secure Coding Standard* [Seacord 2008].

4.2.8 Sample Code

The following sample code canonicalizes a user-supplied pathname before verifying and opening the file.

```
/* Verify argv[1] is supplied */

char *canonical_filename = canonicalize_file_name(argv[1]);
if (canonical_filename == NULL) {
  /* Handle error */
}

/* Verify filename */

if (fopen(canonical_filename, "w") == NULL) {
  /* Handle error */
}

/* ... */

free(canonical_filename);
```

```
canonical_filename = NULL;
```

4.2.9 Known Uses

xarchive-0.2.8-6

4.3 Input Validation

4.3.1 Intent

Many vulnerabilities can be prevented by ensuring that input data is properly validated. Input validation requires that a developer correctly identify and validate all external inputs from untrusted data sources.

4.3.2 Motivation

The use of unvalidated user input by an application is the root cause of many serious security exploits, such as buffer overflow attacks, SQL injection attacks, and cross-site scripting attacks.

Given the prevalence of applications with a client-server architecture, one issue faced by system designers is where to perform the input validation, on the client side or on the server side. Problems in input validation occur when only client-side validation is performed.

Client-side validations are inherently insecure. It is easy to spoof a web page submission and bypass any scripting on the original page. This is more or less true for any type of client-server architecture. However, while you cannot rely on client-side validation, it is still useful. Immediate user feedback can avoid another round trip to the server, saving time and bandwidth.

4.3.3 Example

A university is writing an ERP (Enterprise Resource Planning) application with a web-based interface to allow university employees to enter time sheet information, bill purchases against accounts, and track the status of various funding sources. The university wishes to ensure (among other security considerations) that malicious or incorrect user input does not result in forbidden changes to ERP data, violations of data integrity, or forbidden access to data by a user.

4.3.4 Applicability

This pattern is applicable to any software that accepts data from an untrusted source. Any data that arrives at a program interface across a security boundary requires validation. General examples of such data include `argv`, environment, sockets, pipes, files, signals, shared memory, and devices. Some input sources specific to web applications are `GET` and `POST` parameters from HTTP forms. Other applications may have other input sources.

4.3.5 Structure

The structure of the Input Validation pattern is fairly simple and only requires identifying and validating each untrusted input as shown in Figure 11.

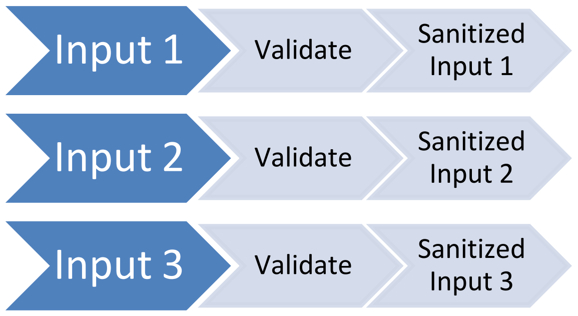

Figure 11: Structure of the Input Validation Pattern

4.3.6 Participants

These are the participants in the Input Validation pattern:

- The system accepting data. The primary participant in this pattern is the system that accepts and validates data.
- External entities providing data. The data provided to the system comes from some external source. Potential data sources include
  - human users
  - files
  - network connections
  - memory shared with other processes
  - database systems

4.3.7 Consequences

The benefits of validating all system input is increased system security (exploits that rely on poor handling of invalid input are prevented) and reliability (the system behaves in a predictable manner when provided with invalid input). The costs of input validation are slower system performance and the additional work required to identify and handle all places where invalid input can occur.

4.3.8 Implementation

The implementation of the Input Validation secure design pattern involves two general design tasks:

- **Specify and validate data.** Data from all untrusted sources must be fully specified and the data validated against these specifications. The system implementation must be designed to handle any range or combination of valid data. Valid data, in this sense, is data that is anticipated by the design and implementation of the system and therefore will not result in the system entering an indeterminate state. For example, if a system accepts two integers as input and multiplies those two values, the system must either (a) validate the input to ensure that an overflow or other exceptional condition cannot occur as a result of the operation or (b) be prepared to handle the result of the operation in the event of an overflow or other exceptional condition. The specifications must address limits, minimum and maximum values, minimum and maximum lengths, valid content, initialization and re-initialization requirements, and encryption requirements for storage and transmission.

- **Ensure that all input meets the specification.** Use data encapsulation (e.g., classes) to define and encapsulate input. For example, instead of checking each character in a user name input to make sure it is a valid character, define a class that encapsulates all operations on that type of input. Input should be validated as soon as possible. Incorrect input is not always malicious; often it is accidental. Reporting the error as soon as possible often helps correct the problem. When an exception occurs deep in the code it is not always apparent that the cause was an invalid input and which input was out of bounds.

A data dictionary or similar mechanism can be used for specification of all program inputs. Input is usually stored in variables, and some input is eventually stored as persistent data. To validate input, specifications for what is valid input must be developed. A good practice is to define data and variable specifications, not just for all variables that hold user input, but also for all variables that hold data from a persistent store. The need to validate user input is obvious; the need to validate data being read from a persistent store is a defense against the possibility that the persistent store has been tampered with.

General Implementation Process

In more detail, the process for implementing this pattern consists of the following steps:

1. **Identify all input sources.** All sources of input to the system must be identified. An input source is any entity or resource that provides data to the system where the received data is non-deterministic; that is, any source of data where the value of the data is not completely determined by the current internal state of the system and past actions performed by the system. As mentioned previously, potential input sources are the file system, a database system, network traffic read via a socket, input from a pipe, the keyboard, etc.
2. **Identify all reads of input sources.** For each input source, identify every point in the system where data from the input source is initially read. Note that if the system has been designed to be loosely coupled from the input sources and hence has interaction with the input sources isolated to a small number of places in the code base, the identification of reads from input sources will be relatively simple. However, if the system was designed so that interaction with data sources is scattered throughout the code base, identification of all reads from input sources will be difficult.
3. **Define criteria for valid data.** For each of the data reads identified in the previous step, define what it means for data read by the current read to be valid. The definition of validity will depend on the type of data being read and what that particular data will be used for. For example:
   a. **Numeric data**. Numeric data should be checked to make sure that it is within some fixed bounds. It should also be checked to ensure it does not cause overflow or underflow errors in subsequent computations. Additional guidance on the checking of numeric data can be found in the CERT C Secure Coding rules and recommendations [Seacord 2008].
   b. **String data**. If the string data is going to be displayed on a web page, it should be sanitized to ensure that it does not contain HTML and client-side script code. If the string

data is going to be used in a database query, it should be sanitized to foil SQL injection attacks.

4. **Figure out how to handle invalid data.** For each of the data reads identified in step two, the behavior of the system when given invalid data should be explicitly defined. Responses to invalid input can range from issuing a warning and continuing with default data to re-requesting the data from the input source. Correct handling of invalid data is a highly application-specific matter.
5. **Add code to check for and handle invalid data.** For each of the data reads identified in step two, code should be written to check the validity of the data read and cases of invalid data should be handled.

   There are two common approaches to identifying invalid data: *blacklisting* and *whitelisting*. Blacklisting consists of comparing input data against a set of inputs known to be invalid, commonly known as a blacklist. If it is not on the blacklist, the input may be considered valid. Whitelisting consists of comparing input data against a set of inputs known to be valid, commonly known as a whitelist. If it is not on the whitelist, the input may be considered invalid. Both whitelisting and blacklisting involve a simple implementation, comparing input against the whitelist or blacklist. The main work comes in maintaining the whitelist or blacklist. When either solution is possible, the whitelist is considered a safer choice because new forms of invalid input need to be entered into a blacklist, but a whitelist requires no change upon discovery of new forms of invalid input.

Additional Implementation Information

Some specific ways to implement input validation in a structured method are available in these sources:

- "Input Validation Using the Strategy Pattern" [Gervasio 2007]. This solution uses the Gang of Four Strategy pattern [Gamma 1995] to handle input validation for various classes of inputs. The presented solution is programmed in PHP.
- "Client/Server Input Validation Using MS ATL Server Libraries" [MSDN 2009c]. This provides an example (in C++) under Windows of doing client-server input validation using input validation routines provided by the ATL libraries.
- *Secure Programming Cookbook* by Viega and Messier [Viega 2003]. This book provides functions and programming strategies for performing input validation in C++.
- "Input Validation in Apache Struts Framework" [You 2009]. This article provides a good tutorial on how to perform input validation when programming in Java using the Apache Struts framework. Of general interest in the tutorial is the detailed specification of valid system input.

4.3.9 Sample Code

This sample code is an example of a structured input validation methodology in C++. Note that there are many other ways to implement the Input Validation pattern.

The basic architecture of the example implementation of the Input Validation pattern is to represent a single set of validation criteria as a `validator` class. A `validator` class is a class with a single static `validate()` method that takes a piece of input to validate and returns `true` if the input is valid and `false` if the input is invalid.

The following `validator` class checks to see if an integer falls within a defined range.

```
template <int lower, int upper> class InRange {

public:

  static bool validate(int item) {
    return ((item >= lower) && (item <= upper));
  }
};
```

The following `validator` class checks to see whether two integers will *not* overflow if multiplied together [Seacord 2008].

```
class NoOverflowOnMult {

public:

  static bool validate(int o1, int o2) {

    // This validation method only works if the size of a long long is
    // greater than double the size of an integer.
    assert(sizeof(long long) >= 2 * sizeof(int));

    signed long long tmp = (signed long long)o1 * (signed long long)o2;

    // If the product cannot be represented as a 32-bit integer,
    // there is overflow.
    return !((tmp > INT_MAX) || (tmp < INT_MIN));
  }
};
```

The following `validator` class checks to see if a string holds a valid name where a valid name contains only alphanumeric characters, contains exactly one space, and is less than a defined number of characters long.

```
template<int maxNameLen> class GoodName {

public:

  static bool validate(char *str) {

    // The name should contain no digits and exactly 1 space.
    unsigned int pos = 0;
    bool sawSpace = false;
    while ((pos < maxNameLen) && (str[pos] != '\0')) {
      // Are we looking at a space in the string we are checking?
      if (str[pos] == ' ') {
        // Is this the 2nd space in the string?
        if (sawSpace) {
          // The name has more than 1 space. It is not a valid name.
          return false;
        }
        // Track that we have seen 1 space.
        sawSpace = true;
      }
      else {
        // Is the current character an alphabetic character?
        if (!isalpha(str[pos])) {
          // The name contains at least 1 non-alphabetic character. It
          // is not a valid name.
          return false;
        }
      }
      // Advance to the next character.
      pos++;
    }

    // A valid name string is less than maxNameLen characters.
    if (str[pos] != '\0') {
      return false;
    }

    // If we get here the name is valid.
    return true;
  }
};
```

The `main()` program provides some examples of how to use the `validator` classes.

```
int main(int argc, const char* argv[]) {
  if (InRange<1,10>::validate(5)) {
    cout << "5 is valid input\n";
  }

  if (!InRange<1,10>::validate(15)) {
    cout << "15 is NOT valid input\n";
  }

  if (NoOverflowOnMult::validate(12, 33)) {
    cout << "12*33 will not overflow\n";
  }

  if (!(NoOverflowOnMult::validate(INT_MAX, 33))) {
    cout << "INT_MAX*33 WILL overflow\n";
  }

  if (GoodName<100>::validate("Corey Duffle")) {
    cout << "'Corey Duffle' is a valid name.\n";
  }

  if (!GoodName<100>::validate("Sir Chumley the 5th")) {
    cout << "'Sir Chumley the 5th' is NOT a valid name.\n";
  }
}
```

4.3.10 Example Resolved

The university has identified three sources of input to their (very simple) ERP system:

- a database system
- GET parameters from HTML forms
- POST parameters from HTML forms

The university has written a utility library to read GET/POST parameters that allows the developers to easily specify a validity checking routine to use when reading GET/POST parameters. The developers are using a simple static analysis tool to ensure that all reads of GET/POST parameters occur only through the utility library. They have instituted formal code reviews of the input validation checking routines to ensure that all input validation criteria are implemented correctly.

The university is using a third-party database abstraction library that sanitizes all provided strings that are to be used in the creation of SQL queries and provides some basic sanity checking of the results of SQL queries.

4.3.11 Known Uses

Many web frameworks and languages and general programming libraries provide support for performing input validation and sanitization. Frameworks with known input validation support include

- Ruby on Rails
- Java Struts
- Pylons
- Django

4.4 Runtime Acquisition Is Initialization (RAII)

4.4.1 Intent

The intent of the RAII pattern is to ensure that system resources are properly allocated and deallocated under all possible program execution paths. RAII ensures that program resources are properly handled by performing resource allocation and deallocation in an object's constructor and destructor, removing the need for external users of an object to handle the allocation and deallocation of the object's resources.

4.4.2 Motivation

Typically every resource that is used must be released in a timely manner. This is necessary to prevent resource exhaustion. It is also important not to release resources while they are still being used. This often has fatal consequences. For instance, usage of memory that has been previously freed is widely considered a security flaw because many memory allocation systems re-use freed memory when further memory is requested. The usage of freed memory might consequently overwrite data that was stored in memory requested after the original memory was freed.

Furthermore, the maintenance of when to free resources often becomes a daunting task due to large numbers of reserved resources. Unless a resource’s lifetime is planned in the software design, it is difficult to determine when the resource is no longer necessary and may be released.

4.4.3 Example

One example of the use of the RAII pattern is a program that allocates memory at the beginning of a function and frees the memory before the function exits. This includes freeing the memory under alternate control flows. For instance, if the function throws an exception or halts the program, it still frees the memory first. Another example of the use of the RAII pattern is an object that opens a network connection when it is constructed and closes the network connection when it is destroyed.

Similarly, an object might open a file when it is constructed. In this case the object must close the file in its destructor. If the opening of the file is optional, the destructor assumes the responsibility for closing the file if and only if it has been opened.

4.4.4 Applicability

RAII applies to any system that uses a resource that must be acquired and subsequently released. Such resources include regions of memory, opened file descriptors, and network resources, such as open sockets.

The pattern is useful when the amount of available resources is finite and limited and when failing to release acquired resources yields resource exhaustion and denial of service.

4.4.5 Structure

The structure of the RAII secure design pattern is relatively straightforward. In the common code executed at the start of the lifetime of an object (commonly in the object’s constructor in object-oriented languages), allocate resources. In the common code executed at the end of the lifetime of an object (commonly the object’s destructor in object-oriented languages), deallocate resources.

4.4.6 Participants

The participants in the RAII pattern are

- the object making use of system resources
- the system resources

4.4.7 Consequences

RAII enforces automatic resource management, in that a resource is acquired only by the object or function that needs it, and the resource is never left unfreed after the object’s lifetime. The program might run more slowly with RAII than it might run with an alternate resource management scheme, such as garbage collection. Such comparisons are highly implementation-dependent.

4.4.8 Implementation

RAII enforces automatic resource management, in that a resource is acquired only by the object or function that needs it. When an object manages a resource, the object typically allocates the resource in its constructor and releases the resource in its destructor. The object's destructor must also be invoked when the object itself is no longer required. But this is itself another instance of RAII, where the object that manages a resource is itself another resource and must be managed by another object or function.

When a function manages a resource, the function typically allocates the resource during its execution (often near the beginning) and releases the resource before it returns. The developer must be aware of all forms of abnormal exit of the function, such as exceptions, and must ensure the resource is released upon any exit venue. That is, if the function calls a subfunction that throws an exception, the function must catch the exception and release the resource before handling the exception or rethrowing it.

For more implementation details, see the following CERT secure coding guidelines:

- For C++, see FIO42-CPP, "Ensure files are properly closed when they are no longer needed for file-based RAII" [CERT 2009b]
- For C, see MEM00-C, "Allocate and free memory in the same module, at the same level of abstraction for memory-based RAII" [Seacord 2008]
- For Java, see FIO34-J, "Ensure all resources are properly closed when they are no longer needed for network-based RAII" [CERT 2009c]

4.4.9 Sample Code

RAII is most prevalent in C++ because an automatic variable object in C++ will have its destructor called when its scope terminates, either normally or through a thrown exception.

The following RAII class is a lightweight wrapper of the C standard library file system calls.

```
#include <cstdio>
#include <stdexcept> // std::runtime_error
class file {
public:
  file (const char* filename) : file_(std::fopen(filename, "w+")) {
    if (!file_)
      throw std::runtime_error("file open failure");
  }

  ~file() {
    if (0 != std::fclose(file_)) { // need to flush latest changes?
      // handle it
    }
  }

  void write (const char* str) {
    if (EOF == std::fputs(str, file_))
      throw std::runtime_error("file write failure");
  }
private:
  std::FILE* file_;

  // prevent copying and assignment; not implemented
  file (const file &);
  file & operator= (const file &);
};
```

Class `file` can then be used as follows:

```
void example_usage() {
  file logfile("logfile.txt"); // open file (acquire resource)
  logfile.write("hello logfile!");
  // continue using logfile ...
  // throw exceptions or return without worrying about closing the
  // log; it is closed automatically when logfile goes out of scope
}
```

This works because the class `file` encapsulates the management of the `FILE*` file handle. When objects `file` are local to a function, C++ guarantees that they are destroyed at the end of the enclosing scope (the function in the example), and the `file` destructor releases the file by calling `std::fclose(file_)`. Furthermore, `file` instances guarantee that a file is available by throwing an exception if the file could not be opened when creating the object.

4.4.10 Known Uses

The BOOST library provides the **`boost::shared_ptr`**, which is also marked for inclusion in the new C++0x standard. It is a smart pointer that uses reference counting to manage pointed-to objects, and it guarantees that the referenced objects are destroyed when the **`shared_ptr`** is destroyed.

5 Conclusion and Future Work

5.1 Conclusion

Secure software development requires secure designs. Secure design patterns can address security issues at varying levels of abstraction. Useful secure design patterns can be created by analyzing and generalizing existing best practices in secure software development that are not immediately identifiable as patterns or by extending existing object-oriented definition design patterns to address security concerns.

While the availability of information and tools to help developers to develop code with fewer security defects has improved over time, information about secure design techniques is not as readily available. Distilling secure design techniques into the context of reusable design patterns allows these techniques to be readily reused. The broader application of secure design should reduce the cost of producing secure products while reducing the risks associated with security vulnerabilities for both developers and end users.

5.2 Future Work

Secure design patterns created by extension from existing object-oriented design patterns have yet to be tested in real world applications. Creating one or more prototypes to evaluate the application of these patterns would be useful in proving their merit and would serve as a reference for developers using them in practice. The use of an object-oriented language such as Java or C++ would be particularly applicable in this case.

Because we were able to extend several traditional object-oriented patterns to add security properties, it would be useful to analyze whether this could be done with other existing non-security related patterns.

Continued mining of existing secure products may identify additional secure design patterns that could be more generally useful.

In the process of describing secure design patterns, a number of techniques that are detrimental to software security can be identified. Specifically documenting these secure design *anti-patterns* can help developers to isolate areas of their software that are at particular risk.

References

URLs are valid as of the publication date of this document. A date in parentheses is a last-updated date, a last accessed date, or a copyright date on a web page.

[Alexander 1977]
Alexander, Christopher. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, 1977 (ISBN 0195019199).

[Beck 1987]
Beck, Kent & Cunningham, Ward. "Using Pattern Languages for Object-Oriented Programs." Design Methodology for Object-Oriented Programming, Panel Session, OOPSLA, 1987. ACM, 1987.

[Bernstein 2008]
Bernstein, Daniel J. The qmail home page. http://cr.yp.to/qmail.html (2008).

[Blakely 2004]
Blakely, Bob, Heath, Craig, et al. *Security Design Patterns*. Berkshire, UK: The Open Group, 2004 (ISBN 1-931624-27-5).

[CERT 2009a]
CERT Secure Coding Standards (March 2009).
https://www.securecoding.cert.org/confluence/x/BgE

[CERT 2009b]
The CERT C++ Secure Coding Standard (March 2009).
https://www.securecoding.cert.org/confluence/x/fQI

[CERT 2009c]
The CERT Sun Microsystems Secure Coding Standard for Java (March 2009).
https://www.securecoding.cert.org/confluence/x/Ux

[Kienzle 2003]
Kienzle, Darrell, Elder, Matthew, Tyree, David, & Edwards-Hewitt, James. "Secure Patterns Repository Version 1.0." November 2003.
http://www.scrypt.net/~celer/securitypatterns/repository.pdf

[Gamma 1995]
Gamma, Erich, Helm, Richard, Johnson, Ralph, & Vlissides, John M. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995 (ISBN 0201633612).

[Gervasio 2007]
Gervasio, Alejandro. "Validating User Input with the Strategy Pattern." Developer Shed, March 6, 2007. http://www.devshed.com/c/a/PHP/Validating-User-Input-with-the-Strategy-Pattern/

[Kernighan 1999]
Kernighan, Brian W. & Pike, Rob. *The Practice of Programming*. Addison-Wesley, 1999 (ISBN 020161586X).

[Massa 2008]
Massa, Jacques. “Migrate an application to Windows Vista, UAC.” BakTek, July 2008. http://www.baktek-web.com/en/topics/UAC.aspx

[MSDN 2009a]
Microsoft Developer Network. “Securable Objects.” MSDN, January 2009. http://msdn.microsoft.com/en-us/library/aa379557.aspx

[MSDN 2009b]
Microsoft Developer Network. “Running with Administrator Privileges.” MSDN, January 2009. http://msdn.microsoft.com/en-us/library/ms717801(VS.85).aspx

[MSDN 2009c]
Microsoft Developer Network. “Input Sample: Demonstrates User Input Validation on Client and Server.” MSDN Visual Studio Developer Center. http://msdn.microsoft.com/en-us/library/x88c4k6b(VS.71).aspx (2009).

[Oppermann 1998]
Oppermann, André. “The big qmail picture.” http://www.nrg4u.com/ (1998).

[Postfix]
The Postfix Home Page, http://www.postfix.org/ (2009).

[Provos 2003]
Provos, Niels. “Privilege Separated OpenSSH.” Center for Information Technology Integration, August 2003. http://www.citi.umich.edu/u/provos/ssh/privsep.html

[Romanosky 2001]
Romanosky, Sasha. “Security Design Patterns, Part 1.” http://www.cgisecurity.com/lib/securityDesignPatterns.html (November 2001).

[Schumacher 2006]
Schumacher, Markus, Fernandez-Buglioni, Eduardo, Hybertson, Duane, Buschmann, Frank, & Sommerlad, Peter. *Security Patterns: Integrating Security and Systems Engineering*. West Sussex, UK: John Wiley and Sons, 2006 (ISBN-10 0-470-85884-2).

[Seacord 2008]
Seacord, Robert. *The CERT C Secure Coding Standard*. Addison-Wesley, 2008.

[Steel 2005]
Steel, Chris, Nagappan, Ramesh, & Lai, Ray. *Core Security Patterns: Best Practices and Strategies for J2EE, Web Services, and Identity Management*. Prentice-Hall, 2005 (ISBN-10 0131463071) .

[Tenouk 2009]
Tenouk. Module H, Windows OS Security, “Access Control Story: Part 1.” http://www.tenouk.com/ModuleH.html (2009).

[Viega 2003]
Viega, John & Messier, Matt. *Secure Programming Cookbook for C and C++: Recipes for Cryptography, Authentication, Input Validation & More*. O’Reilly Media, Inc., 2003 (ISBN-10 0596003943).

[Yoder 1997]
Yoder, Joseph & Barcalow, Jeffrey. “Architectural Patterns for Enabling Application Security.” *Proceedings of the 4th Pattern Languages of Programming Conference*, 1997. http://hillside.net/plop/plop97/Workshops.html

[You 2009]
You, James (Jun B.). “Applying Design Patterns to Your Struts Validation Framework.” Java Boutique. http://javaboutique.internet.com/tutorials/strutsvalid/ (2009).

REPORT DOCUMENTATION PAGE			*Form Approved OMB No. 0704-0188*
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.			
1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE March 2009		3. REPORT TYPE AND DATES COVERED Final
4. TITLE AND SUBTITLE Secure Design Patterns			5. FUNDING NUMBERS FA8721-05-C-0003
6. AUTHOR(S) Chad Dougherty, Kirk Sayre, Robert C. Seacord, David Svoboda, Kazuya Togashi			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213			8. PERFORMING ORGANIZATION REPORT NUMBER CMU/SEI-2009-TR-010
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) HQ ESC/XPK 5 Eglin Street Hanscom AFB, MA 01731-2116			10. SPONSORING/MONITORING AGENCY REPORT NUMBER ESC-TR-2009-010
11. SUPPLEMENTARY NOTES			
12A DISTRIBUTION/AVAILABILITY STATEMENT Unclassified/Unlimited, DTIC, NTIS			12B DISTRIBUTION CODE
13. ABSTRACT (MAXIMUM 200 WORDS) The cost of fixing system vulnerabilities and the risk associated with vulnerabilities after system deployment are high for both developers and end users. While there are a number of best practices available to address the issue of software security vulnerabilities, these practices are often difficult to reuse due to the implementation-specific nature of the best practices. In addition, greater understanding of the root causes of security flaws has led to a greater appreciation of the importance of taking security into account in all phases in the software development life cycle, not just in the implementation and deployment phases. This report describes a set of *secure design patterns*, which are descriptions or templates describing a general solution to a security problem that can be applied in many different situations. Rather than focus on the implementation of specific security mechanisms, the secure design patterns detailed in this report are meant to eliminate the accidental insertion of vulnerabilities into code or to mitigate the consequences of vulnerabilities. The patterns were derived by generalizing existing best security design practices and by extending existing design patterns with security-specific functionality. They are categorized according to their level of abstraction: architecture, design, or implementation.			
14. SUBJECT TERMS Secure design patterns, software security, secure coding			15. NUMBER OF PAGES 70
16. PRICE CODE			
17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UL